Improving Web Server Performance by Caching Dynamic Data

Arun Iyengar and Jim Challenger

IBM Research Division
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
{aruni, challngr}@watson.ibm.com

Abstract

Dynamic Web pages can seriously reduce the performance of Web servers. One technique for improving performance is to cache dynamic Web pages. We have developed the DynamicWeb cache which is particularly well-suited for dynamic pages. Our cache has improved performance significantly at several commercial Web sites. This paper analyzes the design and performance of the DynamicWeb cache. It also presents a model for analyzing overall system performance in the presence of caching. Our cache can satisfy several hundred requests per second. On systems which invoke server programs via CGI, the DynamicWeb cache results in near-optimal performance, where optimal performance is that which would be achieved by a hypothetical cache which consumed no CPU cycles. On a system we tested which invoked server programs via ICAPI which has significantly less overhead than CGI, the DynamicWeb cache resulted in near-optimal performance for many cases and 58% of optimal performance in the worst case. The DynamicWeb cache achieved a hit rate of around 80% when it was deployed to support the official Internet Web site for the 1996 Atlanta Olympic games.

1 Introduction

Web servers provide two types of data: static data from files stored at a server and dynamic data which are constructed by programs that execute at the time a request is made. The presence of dynamic data often slows down Web sites considerably. High-performance Web servers can typically deliver several hundred static files per second. By contrast, the rate at which dynamic pages are delivered is often one or two order of magnitudes slower [10].

One technique for reducing the overhead of dynamic page creation is to cache dynamic pages at the server the first time they are created. That way, subsequent requests for the same dynamic page can access the page from the cache instead of repeatedly invoking a program to generate the same page.

A considerable amount of work has been done in the area of proxy caching. Proxy caches store data at sites that are remote from the server which originally provided the data. Proxy caches reduce network traffic and latency for obtaining Web data because clients can obtain the data from a local proxy cache instead of having to request the data directly from the site providing the data. Although our cache, known as the DynamicWeb cache, can function as a proxy cache, the aspects we shall focus on in this paper are fundamentally different from those of proxy caches. The primary purpose of the DynamicWeb cache is to reduce CPU load on a server which generates dynamic pages and not to reduce network traffic. DynamicWeb is directly managed by the application generating dynamic pages. Although it is not a requirement, DynamicWeb would typically reside on the set of processors which are managing the Web site [3].

Dynamic pages present many complications which is why many proxy servers do not cache them. Dynamic pages often change a lot more frequently than static pages. Therefore, an effective method for invalidating or updating obsolete dynamic pages from caches is essential. Some dynamic pages modify state at the server each time they are invoked and should never be cached.

For many of the applications that use the DynamicWeb cache, it is essential for pages stored in the cache to be current at all times. Determining when dynamic data should be cached and when cached data has become obsolete is too difficult for the Web server to determine automatically. DynamicWeb thus provides API's for Web application programs to explicitly add and delete things from the cache. While this approach complicates the application program somewhat, the performance gains realized by applications deploying our cache have been significant. DynamicWeb has been deployed at numerous IBM and customer Web sites serving a high percentage of dynamic Web pages. We believe that its importance will continue to grow as dynamic content on the Web increases.

1.1 Previous Work

Liu [11] presents a number of techniques for improving Web server performance on dynamic pages including caching and the use of cliettes, which are long-running processes that can hold state and maintain open connections to databases that a Web server can communicate with. Caching is only briefly described. Our paper analyzes caching in considerably more detail than Liu's paper. A number of papers have been published on proxy caching [1, 4, 6, 7, 12, 13, 15, 24]. None of these papers focus on improving performance at servers generating a high percentage of dynamic pages. Gwertzman and Seltzer [8] examine methods for keeping proxy caches updated in situations where the original data are changing. A number of papers have also been published on cache replacement algorithms for World Wide Web caches [2, 18, 22, 23].

2 Cache Design

Our cache architecture is very general and allows an application to manage as many caches as it desires. The application program can choose whatever algorithm it pleases for dividing data among several caches. In addition, the same cache can be used by multiple applications.

Our cache architecture centers around a cache manager which is a long-running daemon process managing storage for one or more caches (Figure 1). Application programs communicate with the cache manager in order to add or delete items from a cache. It is possible to run multiple cache managers concurrently on the same processor by configuring each cache manager to listen for requests on a different port number. A single application can access multiple cache managers. Similarly, multiple applications can access the same cache.

The application program would typically be invoked by a Web server via the Common Gateway Interface (CGI) [21] or a faster mechanism such as the Netscape Server Application Programming Interface (NSAPI) [16], the Microsoft Internet Application Programming Interface (ISAPI) [14], IBM's Internet Connection Application Programming Interface (ICAPI), or Open Market's FastCGI [17]. However, the application does not have to be Web-related. DynamicWeb can be used by other sorts of applications which need to cache data for improved performance. The current set of cache API's are compatible with any POSIX-compliant C or C++ program. Furthermore, the cache is not part of the Web server and can be used in conjunction with any Web server.

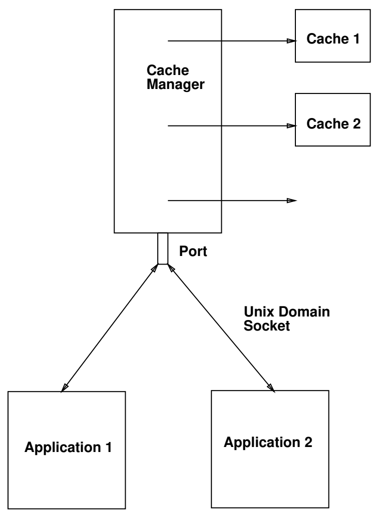

Figure 1: Applications 1 and 2 both have access to the caches managed by the cache manager. The cache manager and both applications are all on the same processor.

The cache manager can exist on a different node from the application accessing the cache (Figure 2). This is particularly useful in systems where multiple nodes are needed to handle the traffic at a Web site. A single cache manager running on a dedicated node can handle requests from multiple Web servers. If a single cache is shared among multiple Web servers, the costs for caching objects is reduced because the object only has to be added to a single cache. In addition, cache updates are simpler, and there are no cache coherency problems.

The cache manager can be configured to store objects in file systems, within memory buffers, or partly within memory and partly within the file system. For small caches, performance is optimized by storing objects in memory. For large caches, some objects have to be stored on disk. The cache

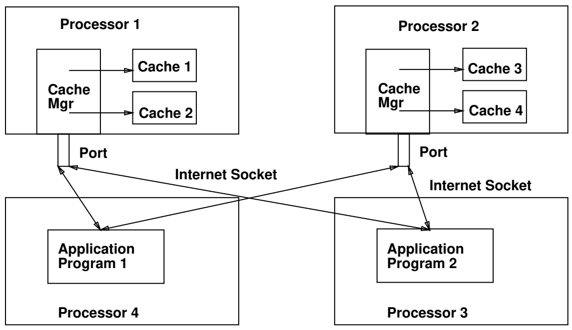

Figure 2: The cache manager and the applications accessing the caches can run on different nodes. In this situation, the cache manager and application communicate over Internet sockets.

manager is multithreaded in order to allow multiple requests to be satisfied concurrently. This feature is essential in keeping the throughput of the cache manager high when requests become blocked because of disk I/O. The cache manager achieves high throughputs via locking primitives which allow concurrent access to many of the cache manager's data structures. When the cache manager and an application reside on different nodes, they communicate via Internet sockets. When the cache manager and an application reside on the same node, they communicate via Unix Domain sockets, which are generally more efficient than Internet sockets.

The overhead for setting up a connection between an application program and a cache can be significant, particularly if the cache resides on a different node than the application program. The cache API's allow long-running connections to be used for communication between a cache manager and an application program. That way, the overhead for establishing a connection need only be incurred once for several cache transactions.

3 Cache Performance

The DynamicWeb cache has been deployed at numerous Web sites by IBM customers. While it has proved to be difficult to obtain reliable performance numbers from our customers, we have extensively measured the performance of the cache on experimental systems at the T. J. Watson Research Center. Section 3.1 presents performance measurements taken from such a system. Section 3.2 presents a method for predicting overall system performance from the performance measurements presented in Section 3.1. Section 3.3 presents cache hit rates which were observed when DynamicWeb was used at a high-volume Web site accessed by people in many different countries.

3.1 Performance Measurements from an Actual System

The system used for generating performance data in this section is shown in Figure 3. Both the cache manager and Web server were on the same node which is an IBM RS/6000 Model 590 workstation running AIX version 4.1.4.0. This machine contains a 66 Mhz POWER2 processor and comprises one node of an SP2 distributed-memory multiprocessor. The Web server was the IBM Internet Connection Secure Server (ICS) version 4.2.1. Three types of experiments were run:

1. Experiments in which requests were made to the cache manager directly from a driver program running on the same node without involving the Web server. The purpose of these experiments was to measure cache performance independently from Web server performance.

2. Experiments in which requests were made to the Web server from remote nodes running the WebStone [19] benchmark without involving the cache. The purpose of these experiments was to measure Web server performance independently of cache performance. WebStone is a widely used benchmark from Silicon Graphics, Inc. which measures the number of requests per second which a Web server can handle by simulating one or more clients and seeing how many requests per second can be satisfied during the duration of the test.

3. Experiments in which server programs which accessed the cache were invoked by requests made to the Web server from remote nodes running WebStone.

The configuration which we used is representative of a high-performance Web site but not optimal. Slightly better performance could probably be achieved by using a faster processor. There are also minor optimizations one can make to the Web server, such as turning off logging, which we didn't make. Such optimizations might have improved performance slightly. However, our goals were to use a consistent set of test conditions so that we could accurately compare the results from different experiments and to obtain good performance

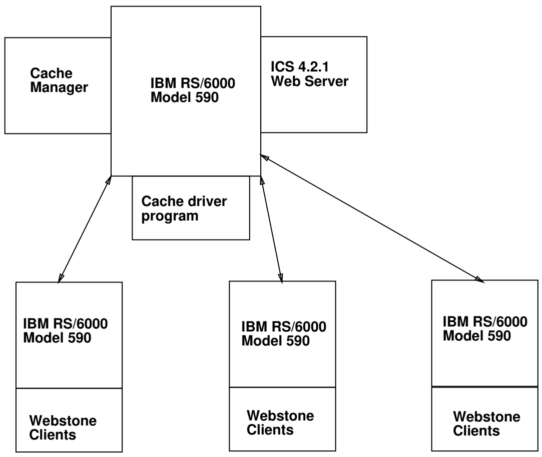

Figure 3: The system used for generating performance data.

but not necessarily the highest throughput numbers possible. Consistent test conditions are crucial, and attempts to compare different Web servers by looking at published performance on benchmarks such as WebStone and SPECweb96 [20] are often misleading because the test conditions will likely differ. Performance is affected by the hardware on which the Web server runs, software (e.g. the operating system, the TCP/IP software), and how the Web server is configured (e.g. whether or not logging is turned on).

As an example of the sensitivity of performance to different test conditions, the Web server and all of the nodes running WebStone (Figure 3) are part of an SP2. The nodes of our SP2 are connected by two networks: an Ethernet and a high-performance switch. The switch has higher bandwidth than the Ethernet. In our case, however, both the switch and the Ethernet had sufficient bandwidth to run our tests without becoming a bottleneck. One would suspect that throughput would be the same regardless of which network was used. However, we observed slightly better performance when the clients running WebStone communicated with the Web server over the switch instead of the Ethernet. This is because the software drivers for the switch are more efficient than the software drivers for the Ethernet, a fact which is unlikely to be known by most SP2 programmers. The WebStone performance numbers presented in this paper were generated using the Ethernet because the switch was frequently down on our system.

Figure 4 compares the throughput of the cache when driven by the driver program to the throughput of the Web server when driven by WebStone running on remote nodes. In both Figures 4 and 5, the cache and Web server were tested independently of each other and did not interact at all. 80% of the requests to the cache manager were read requests and the remaining 20% were write requests. The cache driver program which made requests to the cache and collected performance statistics ran on the same node as the cache and took up some CPU cycles. The cache driver program would not be needed in a real system where cache requests are made by application programs. Without the cache driver program overhead, the maximum throughput would be around 500 requests per second. The cache can sustain about 11% more read requests per second than write requests.

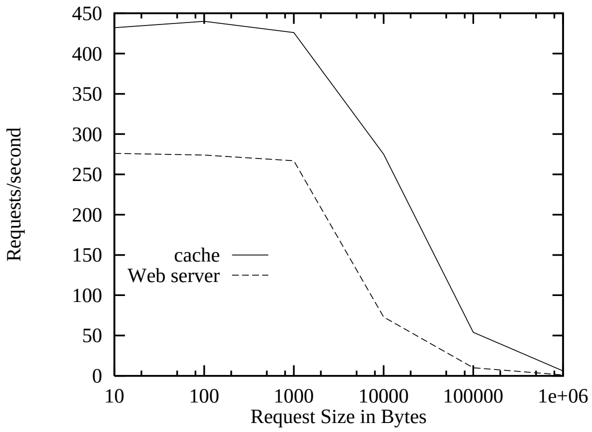

Figure 4: The throughput in requests per second which can be sustained by the cache and the Web server on a single processor. The cache driver program maintained a a single open connection for all requests. Eighty percent of requests to the cache were read requests and 20% were write requests. All requests to the Web server were for static HTML files.

In the experiments summarized in Figure 4, a single connection was opened between the cache driver program and the cache manager and maintained for the duration of the test. A naive interface between the Web server and the cache manager would make a new connection to the cache manager for each request. The first two bars of Figure 5 show the effect of establishing a new connection for each request. The cache manager can sustain close to 430 requests per second when a single open connection is maintained for all requests and about 190 requests per second when a new connection is made for each request. Since the driver program and cache manager were on the same node, Unix domain sockets were used. If they had been on different nodes, Internet sockets would have been needed, and the performance would likely have been worse.

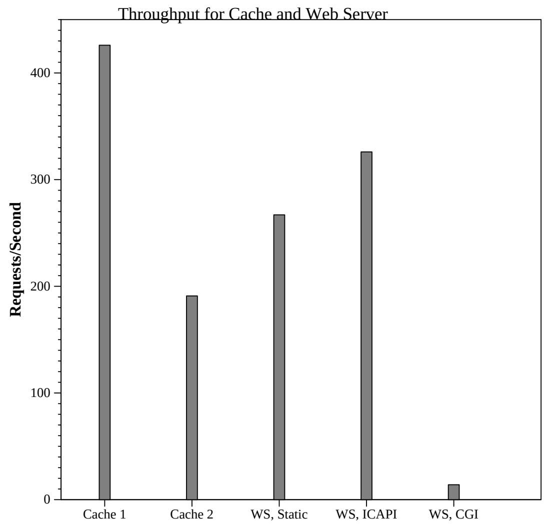

Figure 5: The throughput in requests per second which can be sustained by the cache and the Web server on a single processor under different conditions. The Cache1 bar graph represents the performance of the cache when a single long lived connection is maintained for all requests made by the driver program. The Cache2 bar graph represents the performance of the cache when a new Unix domain socket is opened for each request. The three bar graphs to the right represent the performance of the Web server.

Figure 5 also shows the performance of the Web server for different types of accesses. In both Figures 5 and 6, request sizes were less than 1000 bytes. We saw little variability in performance as a function of request size until request sizes exceeded 1000 bytes (Figure 4). For objects not exceeding 1000 bytes, the Web server can deliver around 270 static files per second. The number of dynamic pages created by very simple programs which can be returned by the ICAPI interface is higher, around 330 per second. The Common Gateway Interface (CGI) is very slow, however. Fewer than 20 dynamic pages per second can be returned by CGI, even if the programs creating the dynamic pages are very simple. The overhead of CGI is largely due to forking off a new process each time a CGI program is invoked.

ICAPI uses a programming model in which the server is multithreaded. Server programs are compiled as shared libraries which are dynamically loaded by the Web server and execute as a thread within the Web server's process. There is thus no overhead for forking off a new process when a server program is invoked through ICAPI. The ICAPI interface is fast. One of the disadvantages to ICAPI, however, is that the server program becomes part of the Web server. It is now much easier for a server program to crash the Web server than if CGI is used. Another problem is that ICAPI programs must be thread-safe. It is not always a straightforward task to convert a legacy CGI program to a thread-safe ICAPI program. Furthermore, debugging ICAPI programs can be quite challenging.

Server API's such as FastCGI use a slightly different programming model. Server programs are long-running processes which the Web server communicates with. Since the server programs are not part of the Web server's process, it is less likely for a server program to crash the Web server compared with the multithreaded approach. FastCGI programs do not have to be thread-safe. One disadvantage is that the FastCGI interface may be slightly slower than the ICAPI one because interprocess communication is required.

Using an interface such as ICAPI, it would be possible to implement our cache manager as part of the Web server which is dynamically loaded as a shared library at the time the Web server is started up. There would be no need for a separate cache manager daemon process. Cache accesses would be faster because the Web server would not have to communicate with a separate process. This kind of implementation is not possible with interfaces such as CGI or FastCGI.

We chose not to implement our cache manager in this fashion because we wanted our cache manager to be compatible with as wide a range of interfaces as possible and not just ICAPI. Another advantage of our design is that it allows the cache to be accessed remotely from many Web servers while the optimized ICAPI approach just described does not.

Figure 6 shows the performance of the Web server when server programs which access the cache are invoked via the ICAPI interface. The first bar shows the throughput when all requests to the cache manager are commented out of the server program. The purpose of this bar is to illustrate the overhead of the server program without the effect of any cache accesses. Slightly over 290 requests/second can be sustained under these circumstances. A comparison of this bar with the fourth bar in Figure 5 reveals that most of the overhead results from the ICAPI interface and not the actual work done by the server program.

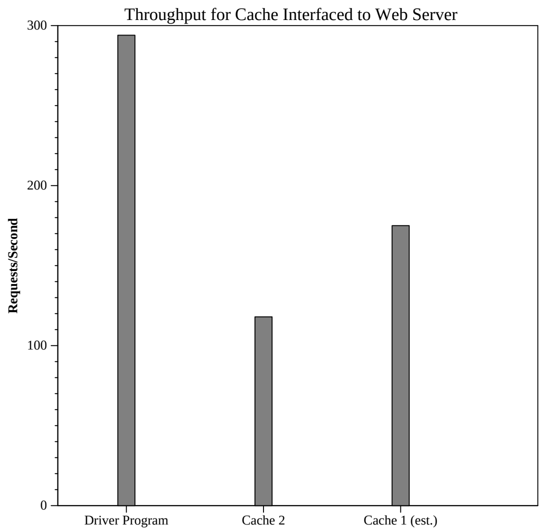

Figure 6: The throughput in requests per second for the cache interfaced to the Web server. The Driver Program bar graph is the throughput which can be sustained by the Web server running the cache driver program through the ICAPI interface with the calls to the cache manager commented out. The Cache 2 bar graph is the throughput for the same system with the cache manager calls. Each cache request from the Web server opens a new connection to the cache manager. The Cache 1 bar graph is an estimate of the throughput of the entire system if the Web server were to maintain long-lived open connections to the cache manager.

The second bar shows the performance of the Web server when each server program returns an item of 1000 bytes or less from the cache. Each request opens up a new connection with the cache manager. About 120 requests/second can be sustained. The observed performance is almost exactly what one would calculate by combining the average request time of the cache (represented by the reciprocal of the second bar in Figure 5) and the Web server driver program (represented by the reciprocal of the first bar in Figure 6) both measured independently of each other.

Performance can be improved by maintaining persistent open connections between the cache manager and Web server. That way, new connections don't have to be opened for each request. The performance one would expect to see under these circumstances is shown in the third bar of Figure 6. It is obtained by combining the average request time of the cache (represented by the reciprocal of the first bar in Figure 5) and the Web server driver program (represented by the reciprocal of the first bar in Figure 6) both measured independently of each other. The reciprocal of this quantity is the throughput of the entire system which is 175 requests/second.

3.2 An Analysis of System Performance

The throughput achieved by any system is limited by the overhead of the server program which communicates with the cache. If server programs are invoked via CGI, this overhead is generally over 20 times more than the CPU time for the cache manager to perform a single transaction. The result is that the cache manager consumes only a small fraction of the CPU time. Using a faster cache than the DynamicWeb cache would have little if any impact on overall system performance. In other words, the DynamicWeb cache results in near-optimal performance.

When faster interfaces for invoking server programs are used, the CPU time consumed by the DynamicWeb cache becomes more significant. This section presents a mathematical model of the overall performance of a system similar to the one we tested in the previous section in which server programs are invoked through ICAPI, which consumes much less CPU time than CGI. The model demonstrates that DynamicWeb achieves near-optimal system throughput in many cases. In the worst case, DynamicWeb still manages to achieve 58% of the optimal system throughput.

Consider a system containing a single processor running both a Web server and one or more cache managers. Let us assume that the performance of the system is limited by the processor's CPU. Define

h = cache hit rate expressed as the proportion of requests which can be satisfied from the cache.

s = average CPU time to generate a dynamic page by invoking a server program (i.e. CPU time for a cache miss).

c = average CPU time to satisfy a request from the cache (i.e. CPU time for a cache hit). $c = c' + c''$ where c' is the average CPU time taken up by a program invoked by the Web server for communicating with a cache manager and c'' is the average CPU time taken up by a cache manager for satisfying a request.

p_{dyn} = proportion of requests for dynamic pages

f = average CPU time to satisfy a request for a static file.

Then the average CPU time to satisfy a request on the system is:

$$T = (h * c + (1-h) * s) * p_{dyn} + f * (1 - p_{dyn}) \quad (1)$$

System performance is often expressed as *throughput* which is the number of requests which can be satisfied per unit time. Throughput is the reciprocal of the average time to satisfy a request. The throughput of the system is given by:

$$T_{tp} = \frac{1}{(\frac{h}{c_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}}} \quad (2)$$

where $T_{tp} = 1/T$, $c_{tp} = 1/c$, $s_{tp} = 1/s$, and $f_{tp} = 1/f$.

The number of requests per second which can be serviced from our cache manager, c_{tp}, is around 175 per second in the best case on the system we tested. Most of the overhead in such a situation results from c' because invoking server programs is costly, even using interfaces such as NSAPI and ICAPI.

The number of dynamic pages per second which can be generated by a server program, s_{tp}, varies considerably depending on the application. Values for s_{tp} as low as 1 per second are not uncommon. The overhead of the Common Gateway Interface (CGI) is enough to limit s_{tp} to a maximum of around 20 per second for any server program using this interface. In order to get higher values of s_{tp}, an interface such as NSAPI, ISAPI, ICAPI, or FastCGI must be used instead.

The rate at which static files can be served, f_{tp}, is typically several hundred per second on a high-performance system. On the system we tested, f_{tp} was around 270 per second. The proportion of requests for dynamic pages, p_{dyn}, is typically less than .5, even for Web sites where all hypertext links are dynamic. This is because many dynamic pages at such Web sites include one or more static image files.

Figures 7 shows the system throughput T_{tp} which can be achieved by a system similar to ours when all of the requests are for dynamic pages. The parameter values used by this and all other graphs in this section were obtained from the system we tested and include c_{tp} = 175 requests per second and f_{tp} = 270 requests per second. Two curves are shown for nonzero hit rates, one representing optimal T_{tp} values which would be achieved by a hypothetical system where the cache manager didn't consume any CPU cycles and another representing T_{tp} values which would be achieved by a cache similar to ours.

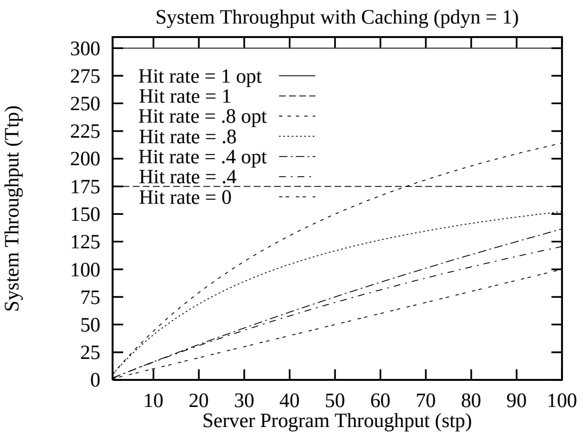

Figure 7: The throughput in connections per second (T_{cp}) achieved by a system similar to ours when all requests are for dynamic pages. The curves with legends ending in *opt* represent hypothetical optimal systems in which the cache manager consumes no CPU cycles.

Figures 8 and 9 are analogous to Figure 7 when the proportion of dynamic pages are .5 and .2 respectively. Even Figure 9 represents a very high percentage of dynamic pages. Web sites for which almost all hypertext links are dynamic could have p_{dyn} close to .2 because of static image files embedded within dynamic pages. The official Internet Web site for the 1996 Atlanta Olympic Games (Section 3.3) is such as an example.

These graphs show that DynamicWeb often results in near optimal system throughput, particularly when the cost for generating dynamic pages is high (i.e. s_{tp} is low). This is precisely the situation when caching is essential for improved performance. In the worst case, DynamicWeb manages to achieve 58% of the optimal system performance.

Another important quantity is the *speedup,* which is the throughput of the system with caching divided by the throughput of the system without caching:

$$S = \frac{\frac{p_{dyn}}{s_{tp}} + \frac{1-p_{dyn}}{f_{tp}}}{(\frac{h}{c_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}}} \quad (3)$$

Figure 10 shows the speedup S which can be achieved by a system similar to ours when all of

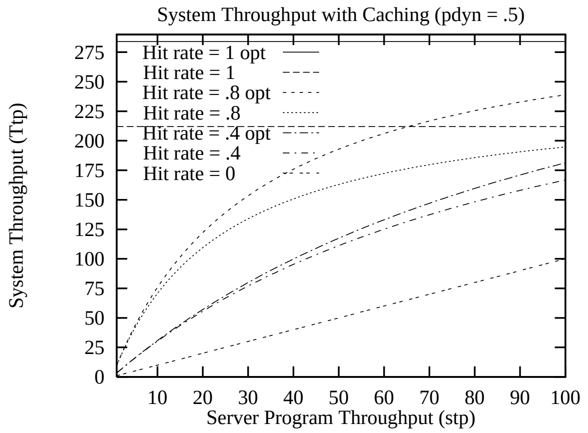

Figure 8: The throughput in connections per second (T_{cp}) achieved by a system similar to ours when 50% of requests are for dynamic pages.

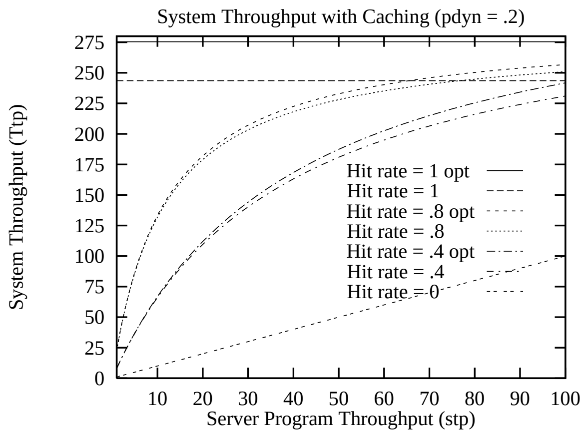

Figure 9: The throughput in connections per second (T_{cp}) achieved by a system similar to ours when 20% of requests are for dynamic pages.

the requests are for dynamic pages. Figures 11 and 12 are analogous to Figure 10 when the proportion of dynamic pages are .5 and .2 respectively. For hit rates below one, DynamicWeb achieves near optimal speedup when the cost for generating dynamic pages is high (i.e. s_{tp} is low). Furthermore, for any hit rate below 1, there is a maximum speedup which can be achieved regardless of how low s_{tp} is. This behavior is an example of Amdahl's Law [9]. The maximum speedup which can be achieved for a given hit rate is independent of the proportion of dynamic pages, p_{dyn}. However, for identical values of s_{tp}, the speedup achieved for a high value of p_{dyn} is greater than the speedup achieved for a lower value of p_{dyn} although this difference approaches 0 as s_{tp} approaches 0.

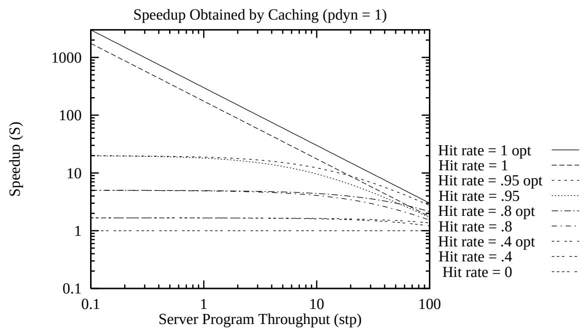

Figure 10: The speedup S achieved by a system similar to ours when all requests are for dynamic pages. The curves with legends ending in *opt* represent hypothetical optimal systems in which the cache manager consumes no CPU cycles.

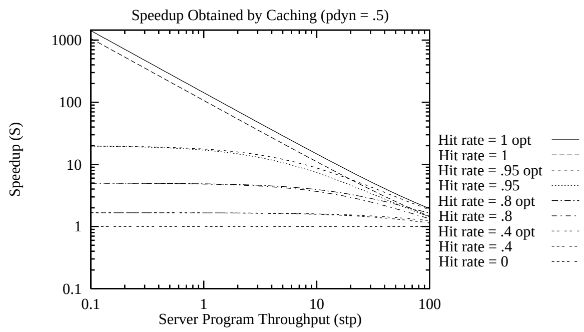

Figure 11: The speedup S achieved by a system similar to ours when 50% of requests are for dynamic pages.

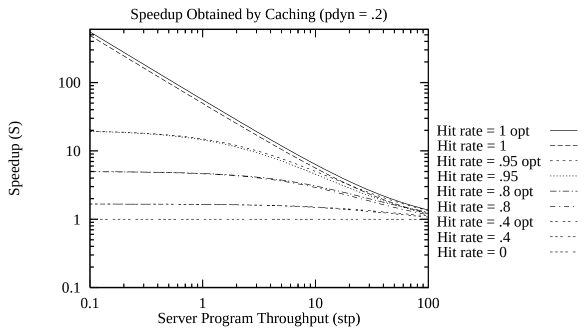

Figure 12: The speedup S achieved by a system similar to ours when 20% of requests are for dynamic pages.

3.2.1 Remote Shared Caches

In some cases, it is desirable to run the cache manager on a separate node from the Web server. An example of this situation would be a multiprocessor Web server where multiple processors each running one or more Web servers are needed to service a high-volume Web site [5]. A cache manager running on a single processor has the throughput to satisfy requests from several remote Web server nodes. One advantage to using a single cache manager in this situation is that cached data only needs to be placed in one cache. The overhead for caching new objects or updating old objects in the cache is reduced. Another advantage is that there is no need to maintain coherence among multiple caches distributed among different processors.

We need a modified version of Equation 2 to calculate the throughput of each Web server in this situation. Recall that c' is the average CPU time taken up by a program invoked by the Web server for communicating with a cache manager. Let $c'_{tp} = 1/c'$. When CGI is used, c'_{tp} is around 20 per second. Most of the overhead results from forking off a new process for each server program which is invoked. When ICAPI is used, c'_{tp} is around 300 per second, and the overhead resulting in c' is mostly due to the ICAPI interface, not the work done by the server programs. The throughput each Web server can achieve is

$$T'_{tp} = \frac{1}{(\frac{h}{c'_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}}} \quad (4)$$

The right hand side of this equation is the same as that for Equation 2 except for the fact that c_{tp} has been replaced by c'_{tp}.

In a well-designed cache such as ours, cache misses use up few CPU cycles. The vast majority of cache manager cycles are consumed by cache hits. The throughput of cache hits for a Web server running at 100% capacity is given by

$$H_n = T'_{tp} * p_{dyn} * h = \frac{p_{dyn} * h}{(\frac{h}{c'_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}}} \quad (5)$$

The number of nodes running Web servers that a single node running a DynamicWeb cache manager can service without becoming a bottleneck when all Web server nodes are running at 100% capacity is

$$N = \frac{c''_{tp}}{H_n} = \frac{c''_{tp} * ((\frac{h}{c'_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}})}{p_{dyn} * h} \quad (6)$$

where $c''_{tp} = 1/c''$ (recall that c'' is the average CPU time taken by a cache manager for satisfying a request). For our system, c''_{tp} is close to 500 requests per second.

Figure 13 shows the number of nodes running Web servers that a single node running a DynamicWeb cache manager can service without becoming a bottleneck when Web server programs are invoked via CGI. It is assumed that the proportion of all Web requests for dynamic pages is .2, the Web server has performance similar to the performance of ICS 4.2.1, and the nodes in the system have performance similar to that of the IBM RS/6000 Model 590. It is also assumed that Web server nodes are completely dedicated to serving Web pages and are running at 100% capacity. If Web server nodes are performing other functions in addition to serving Web pages or are running at less than 100% capacity, the number of Web server nodes which can be supported by a single cache node increases. Figure 14 shows the analogous graph when Web server programs are invoked via ICAPI. Since ICAPI is much more efficient than CGI, Web server nodes can handle more requests per unit time and thus make more requests on the cache manager. The net result is that the cache node can support fewer Web server nodes before becoming a bottleneck.

3.3 Cache Hit Rates at a High-Volume Web Site

DynamicWeb was used to support the official Internet Web site for the 1996 Atlanta Olympic Games. This Web site received a high volume of requests from people all over the world. In order to handle the huge volume of requests which were received, several processors were utilized to provide results to the public. Each processor contained a

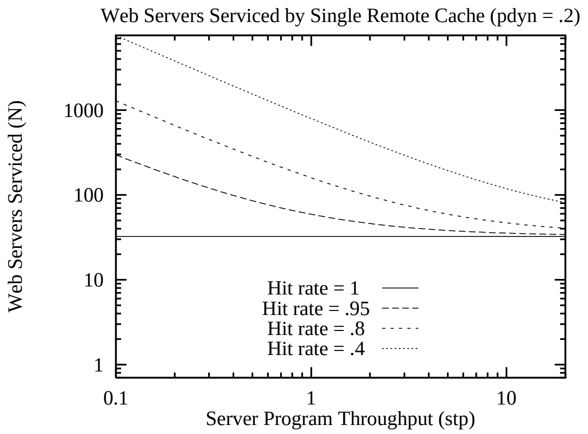

Figure 13: The number of remote Web server nodes that a single node running a DynamicWeb cache manager can service before becoming a bottleneck when Web server programs are invoked via CGI. Twenty percent of requests are for dynamic pages. Due to the overhead of CGI, s_{tp} cannot exceed 20 requests/second which is how far the X-axis extends.

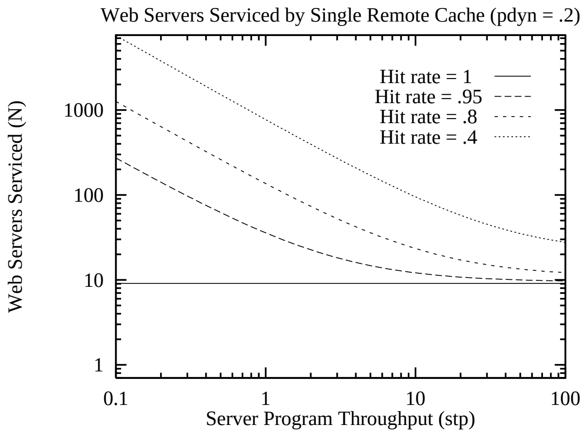

Figure 14: This graph is analogous to the one in Figure 13 when Web server programs are invoked via ICAPI.

Web server, IBM's DB2 database, and a cache manager which managed several caches. Almost all of the Web pages providing Olympics results were dynamically generated by accessing DB2. The proportion of Web server requests for dynamic pages was around .2. The remaining requests were mostly for image files embedded within the dynamic pages. Caching reduced server load considerably; the average CPU time to satisfy requests from a cache was about two orders of magnitude less than the average CPU time to satisfy requests by creating a new dynamic page.

Each cache manager managed 37 caches. Thirty-four caches were for specific sports such as badminton, baseball, and basketball. The remaining three caches were for medal standings, athletes, and schedules. Partitioning pages among multiple caches facilitated updates. When new results for a particular sport such as basketball were received by the system, each cache manager would invalidate all pages from the basketball cache without disturbing any other caches.

In order to optimize performance, cache performance monitoring was turned off for most of the Olympics. Table 1 shows the hit rates which were achieved by one of the servers when performance monitoring was enabled for a period of 2 days, 7 hours, and 40 minutes starting at 12:25 PM on July 30. The average number of read requests per second received by the cache manager during this period was just above 1.

The average cache hit rate for this period was .81. Hit rates for individual caches ranged from a high of .99 for the medal standings cache to a low of .28 for the athletes cache. Low hit rates in a cache were usually caused by frequent updates which made cached pages obsolete. Whenever the system was notified of changes which might make any pages in a cache obsolete, all pages in the cache were invalidated. A system which invalidated cached Web pages at a smaller level of granularity should have been able to achieve a better overall hit rate than .81. Since the Atlanta Olympics, we have made considerable progress in improving hit rates by minimizing the number of cached pages which need to be invalidated after a database update.

In all cases, the servers contained enough memory to store the contents of all caches with room to spare. Cache replacement policies were not an issue because there was no need to delete an object which was known to be current from a cache. Objects were only deleted if they were suspected of being obsolete.

Cache Name	*Read Requests*	*Hits*	*Hit Rate*	*Request Proportion*
Athletics	34216	25385	.74	.165
Medals	17334	17116	.99	.084
Badminton	13479	12739	.95	.065
Table Tennis	12111	11176	.92	.058
Athletes	12009	3415	.28	.058
All 37 Caches	207117	167859	.81	1.000

Table 1: Cache hit rates for the five most frequently accessed caches and all 37 caches combined. The rightmost column is the proportion of total read requests directed to a particular cache. The *Athletics* cache includes track and field sports. The *Medals* cache had the highest hit rate of all 37 caches while the *Athletes* cache had the lowest hit rate of all 37 caches.

4 Conclusion

This paper has analyzed the design and performance of the DynamicWeb cache for dynamic Web pages. DynamicWeb is better suited to dynamic Web pages than most proxy caches because it allows the application program to explicitly cache, invalidate, and update objects. The application program can ensure that the cache is up to date. DynamicWeb has significantly improved the performance of several commercial Web sites providing a high percentage of dynamic content. It is compatible with all commonly used Web servers and all commonly used interfaces for invoking server programs.

On an IBM RS/6000 Model 590 workstation with a 66 Mhz POWER2 processor, DynamicWeb could satisfy close to 500 requests/second when it had exclusive use of the CPU. On systems which invoke server programs via CGI, the DynamicWeb cache results in near-optimal performance, where optimal performance is that which would be achieved by a hypothetical cache which consumed no CPU cycles. On a system we tested in which Web servers invoked server programs via ICAPI which has significantly less overhead than CGI, the DynamicWeb cache resulted in near-optimal performance in many cases and 58% of optimal performance in the worst case. The DynamicWeb cache achieved a hit rate of around 80% when it was deployed to support the official Internet Web site for the 1996 Atlanta Olympic games.

5 Acknowledgments

Many of the ideas for the DynamicWeb cache came from Paul Dantzig. Yew-Huey Liu, Russell Miller, and Gerald Spivak also made valuable contributions.

References

[1] M. Abrams et al. Caching Proxies: Limitations and Potentials. In *Fourth International World Wide Web Conference Proceedings*, pages 119–133, December 1995.

[2] J. Bolot and P. Hoschka. Performance Engineering of the World Wide Web: Application to Dimensioning and Cache Design. *World Wide Web Journal*, pages 185–195, 1997.

[3] J. Challenger and A. Iyengar. Distributed Cache Manager and API. Technical Report RC 21004, IBM Research Division, Yorktown Heights, NY, October 1997.

[4] A. Chankhunthod et al. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, pages 153–163, January 1996.

[5] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, February 1996.

[6] C. Dodge, B. Marx, and H. Pfeiffenberger. Web cataloging through cache exploitation and steps toward Consistency Maintenance. *Computer Networks and ISDN Systems*, 27:1003–1008, 1995.

[7] S. Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27:165–173, 1994.

[8] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, pages 141–151, January 1996.

[9] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, second edition, 1996.

[10] A. Iyengar, E. MacNair, and T. Nguyen. An Analysis of Web Server Performance. In *Proceedings of GLOBECOM '97*, November 1997.

[11] Y. H. Liu, P. Dantzig, C. E. Wu, J. Challenger, and L. M. Ni. A Distributed Web Server and its Performance Analysis on Multiple Platforms. In *Proceedings of the International Conference for Distributed Computing Systems*, May 1996.

[12] A. Luotonen and K. Altis. World Wide Web proxies. *Computer Networks and ISDN Systems*, 27:147–154, 1994.

[13] R. Malpani, J. Lorch, and D. Berger. Making World Wide Web Caching Servers Cooperate. In *Fourth International World Wide Web Conference Proceedings*, pages 107–117, December 1995.

[14] Microsoft Corporation. (ISAPI) Overview. http://www.microsoft.com/msdn/sdk/plat forms/doc/sdk/internet/src/isapimrg.htm.

[15] M. Nabeshima. The Japan Cache Project: An Experiment on Domain Cache. In *Sixth International World Wide Web Conference Proceedings*, 1997.

[16] Netscape Communications Corporation. The Server-Application Function and Netscape Server API. http://www.netscape.com/newsref/std/server_api.html.

[17] Open Market. FastCGI. http://www.fastcgi.com/.

[18] P. Scheuermann, J. Shim, and R. Vingralek. A Case for Delay-Conscious Caching of Web Documents. In *Sixth International World Wide Web Conference Proceedings*, 1997.

[19] Silicon Graphics, Inc.. World Wide Web Server Benchmarking. http://www.sgi.com/Products/WebFORCE/WebStone/.

[20] System Performance Evaluation Cooperative (SPEC). SPECweb96 Benchmark. http://www.specbench.org/osg/web96/.

[21] Various. Information on CGI. hoohoo.ncsa.uiuc.edu:80/cgi/overview.html, www.yahoo.com/Computers/World_Wide_Web/CGI__Common_Gateway_Interface/, www.stars.com/, and www.w3.org/pub/WWW/CGI.

[22] S. Williams et al. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of SIGCOMM '96*, pages 293–305, 1996.

[23] R. P. Wooster and M. Abrams. Proxy Caching That Estimates Page Load Delays. In *Sixth International World Wide Web Conference Proceedings*, 1997.

[24] A. Yoshida. MOWS: Distributed Web and Cache Server in Java. In *Sixth International World Wide Web Conference Proceedings*, 1997.