Integrating Code and Fragment Caching to Speedup Dynamic Web-Page Construction

Suresha
Database Systems Lab, SERC/CSA
Indian Institute of Science
Bangalore 560012, INDIA

Jayant R. Haritsa
Database Systems Lab, SERC/CSA
Indian Institute of Science
Bangalore 560012, INDIA

Abstract

Many web-sites incorporate dynamic web-pages in order to deliver customized contents to their users. However, these dynamic pages, due to their construction overheads, result in substantially increased user response times and server load. In this paper, we consider mechanisms for reducing these overheads by integrating two caching techniques – fragment-caching and code-caching. The experimental results from a detailed simulation study of our techniques indicate that, given a fixed cache budget, the proposed integrated caching performs significantly better than the caching techniques in isolation. We also consider augmenting integrated caching with anticipatory page pre-generation in order to deliver dynamic web-pages faster during normal operating situations, by utilizing the excess capacity with which web-servers are typically provisioned.

1 Introduction

To deliver customized contents to their users, web-sites are increasingly shifting from a static web-page service model to a *dynamic* model [6]. Dynamic web-pages enable much richer interactions than static pages, but these benefits are obtained at the cost of significantly increased user response times, due to the on-demand page construction. Dynamic web-pages also adversely impact the web-server performance due to the extra load incurred by the page generation process. In fact, it has been recently estimated that server-side latency accounts for as much as 40 percent of the total page delivery time experienced by end-users [12]. Hence, performance and scalability are becoming major issues for dynamic web-sites.

International Conference on Management of Data
COMAD 2005b, Hyderabad, India, December 20–22, 2005
©Computer Society of India, 2005

Fragment and Code Caching. To address these issues, a variety of optimization techniques have been developed in the recent literature. These include client-side prefetching, dynamic content-aware full-page caching, database caching, content acceleration, fragment-caching, and code caching [5, 6, 9, 13, 16, 17, 25]. Among these techniques, *fragment-caching*, and the more recent *code-caching*, are particularly attractive. Specifically, fragment-caching reduces dynamic page construction time by caching dynamic fragments, with the following desirable guarantees [5, 6]: Firstly, it ensures the *freshness* of the page contents by maintaining an association between the cached dynamic fragments and the underlying data sources. Secondly, it ensures the *correctness* of the page contents by newly generating the page skeleton each time the dynamic page is requested.

In contrast to fragment-caching, code-caching does not cache results, but caches the compiled code that leads to results. The benefit of code caching is that whenever there is a request for a script execution, the time-consuming script parsing and compilation is bypassed since the compiled code for the script is already available. Moreover, code-caching can work independent of other middle-ware solutions such as fragment-caching.

Page Pre-generation. While caching can certainly address part of the server latency problem, yet it is only half-the-story, because of the following reasons: The utility of fragment-caching is predicated on having a significant portion of dynamic fragments to be cacheable – however, such cacheability may not always be found in practice. Further, even when most fragments are cacheable, dynamic page construction is begun only after receiving the request for the page. Similarly, code-caching focuses on reducing execution time *after* a request is received, but does not avoid the execution itself.

Therefore, it is attractive to consider the additional possibility of resorting to dynamic page *pre-generation*, in conjunction with the caching techniques. Pre-generation is based on having a statistical prediction mechanism for estimating the next page that would be accessed by a user during a session. The page pre-generation is executed during the time period between sending out the response to the user's current request and the receipt of her subsequent request. Note that in the case where the page prediction turns out to be right, the pre-generation effectively reduces the server latency to *zero*, which is the best that could be hoped for from the user perspective.

An unsuccessful pre-generation on the other hand represents wasted effort on the part of the server. This may not be an issue for web-servers that are under normal operation since these systems are usually over-provisioned in order to handle peak loads [10, 19, 21], and therefore some wastage of the excess capacity is not of consequence. But, during peak loads, the additional effort may further exacerbate the system performance.

Cache Partitioning. A related design issue is that space has to be allocated in the server cache to store fragments, compiled code, and pre-generated pages. That is, the cache has to be *partitioned* into these three sub-caches, and the relative sizings of these partitions has to be determined.

1.1 Contributions

We investigate in this paper the possibility of achieving significant reductions in server latencies, and thereby user response times, by a combination of fragment-caching and code-caching, optionally augmented with anticipatory page pre-generation. It represents the next logical step from our previous work [23], which considered the combination of fragment-caching and page-pregeneration.

Our approach ensures the *freshness* of content through either fresh fragment computation or by accessing fragments from the fragment-cache, and the *correctness* of the page contents by newly generating the page skeleton each time the dynamic web-page is requested. Overall, our goal is to achieve *long-term benefits through integrated fragment and code-caching*, and *immediate benefits through anticipatory page pre-generation*.

Using a detailed simulation model of a dynamic web-server, we study the performance of our integrated-caching approach in terms of reducing dynamic web-page construction times, as compared to pure fragment-caching and pure code-caching approaches. Our evaluation is conducted over a range of fragment-caching levels for a given cache budget. The results show that the integrated-caching approach is able to achieve significantly better reductions in server latency as compared to the pure fragment-caching and pure code-caching approaches. Further, for the workloads evaluated in our study, we demonstrate that, given a fixed cache budget, simple heuristics exist for determining close-to-ideal sizings of the cache partitions for fragments and compiled codes.

Our experimental results also show that the combination of integrated-caching with anticipatory page pre-generation reduces response times even further during normal loading. Moreover, by augmenting our system with a simple load-thresholding feedback mechanism, we are able to achieve comparable performance during peak loading.

1.2 Organization

The remainder of this paper is organized as follows: In Section 2, we discuss our integrated fragment and code-caching technique. The incorporation of page pre-generation along with a load-thresholding feedback system is discussed in Section 3. The simulation model for evaluating the various alternatives is described in Section 4, and the experimental results are highlighted in Section 5. Related work is reviewed in Section 6. Finally, in Section 7, we summarize our contributions and outline future research avenues.

2 Integrated Fragment and Code Caching

In this section, we describe in detail our proposed integrated caching architecture. Before discussing our new approach, we first provide background material on fragment-caching and code-caching techniques.

2.1 Fragment Caching

When a web-site receives a request for a dynamic web-page, it has to execute a script corresponding to the request. A script is essentially a set of executable code blocks. Each such code block carries out some computation to generate a part of the required page, and results in an HTML fragment. An output statement after the code block places the resulting HTML fragment in a buffer. Once all executable code blocks in a script have been executed, their output is assembled together, along with the static component, and the resulting HTML is sent as a page to the user.

An executable code block can be tagged as cacheable, if its output is known to not change for a sustained period of time. When the script is executed, these tags instruct the application server to first search for the fragment in the fragment-cache. If the search is successful, the code block execution is bypassed and the content is returned from the cache. If not, the code block is executed and the fragment is generated freshly and also cached for future benefit. The cache contents are managed by an invalidation mechanism and a cache replacement policy. A cached fragment is invalidated whenever the underlying data source updates the data values on which the fragment is dependent. The details of fragment caching are available in [5, 6] – we assume the use of their techniques in the rest of this paper.

2.2 Code Caching

This technique has been recently proposed and implemented in the context of the PHP Accelerator project [16]. Before executing a script, the engine reads, parses, and compiles the contents into code ready for execution. Since, in practice, the scripts rarely change, this pre-processing is targeted for elimination in [16]. They assume that the compiled code uses instructions from a virtual instruction set that is platform independent and once compiled, the code is executed by a virtual machine that interprets the instructions. In our context, it is also possible to conceive that since dynamic web-sites are usually meant for a specific task, the script can be directly compiled to the native machine, dispensing with platform independence. This compiled code can also be optimized before being put into execution to improve its efficiency and reduce its footprint.

High performance can be achieved by using a shared memory cache from which the compiled code can be executed directly. We assume that the code caching technique can be extended to the level of individual code blocks, where, instead of caching the entire compiled code of a script, only the compiled code corresponding to an executable code block is cached.

2.3 Integrated Caching

Our proposed integrated caching model is a simple combination of fragment-caching and code-caching. A high level representation of the proposed integrated caching architecture is given in Figure 1.

Here, a request for a dynamic page triggers the execution of the corresponding script. While executing the script, for all those fragments of the script for which the outputs are already available in the fragment cache, the execution of the fragment is bypassed, and we save on the fragment execution time. On the other hand, for those fragments for which either the outputs are currently not available in the fragment-cache, or the outputs are invalid, or the fragments themselves are not cacheable, the fragment code must be executed from scratch.

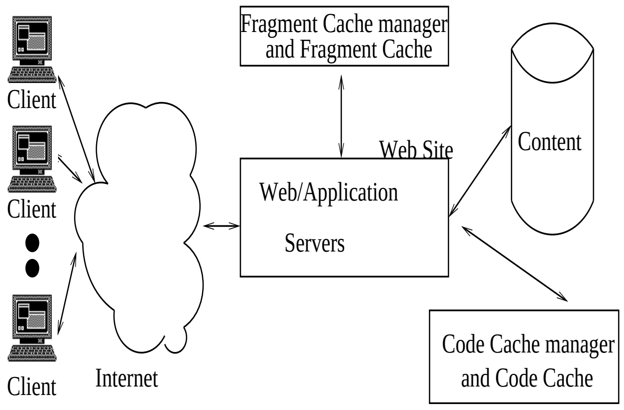
While executing the fragment code, the web-server must first interpret and compile the fragment script and then execute it. In our proposed integrated caching approach, we cache the compiled fragment code in the code-cache, so that any future request for the execution of the same fragment code will save on the compilation overheads, if the compiled fragment code is still available in the cache.

In principle, it is possible to cache both the output and the code of a cacheable fragment, but this appears to be an overkill since if the output is long-lived, then its associated code will be accessed only rarely. Therefore, it appears better to cache only the codes of *uncacheable* fragments, since these are the codes that will be frequently utilized.

It is important to note that the above solution is guaranteed to serve *fresh* content, since it is associated with the origin server. Moreover, it also ensures serving *correct* pages, since the page is specific to the user request. From a broad perspective, by integrated-caching we are achieving the long-term benefits whenever the fragments or their associated compiled codes are reused in course of time.

2.4 Server Cache Management

In our integrated caching approach, we need to allocate space in the server cache for hosting both fragments and compiled fragment codes. Therefore, we partition the cache into a *fragment-cache* and a *code-cache*.



Client
Client
Client
Internet
Fragment Cache manager and Fragment Cache
Web Site
Web/Application Servers
Content
Code Cache manager and Code Cache

Figure 1: Integrated Caching Model

2.4.1 Cache Partition Sizing

An immediate issue that arises here is determining the relative sizes of the fragment-cache and code-cache partitions. This issue is investigated in detail in our experimental study presented in Section 5 – our results there indicate that the size of the code-cache must be in accordance with Equation 1, given in Section 5.3.

2.4.2 Cache Replacement Policies

Apart from sizing, we also need to decide on cache replacement policies. With regard to the fragment-cache, we are not aware of any web logs that are available to track the reference patterns for fragment access. This restricts us to the use of simple techniques like Least Recently Used (LRU) for managing the fragment-cache. A similar technique can be applied for code-caching as well.

An association between the fragments in the fragment-cache and the data in the origin server is maintained. Whenever a fragment is invalidated, it is marked invalid in the fragment-cache, so that the further use of such a fragment is avoided.

In general, no such association is required for the compiled codes in the code-cache since their source scripts rarely change, and further, these changes are usually made manually, in which case the system administrator can forcibly flush the code-cache. But in an environment where script changes are frequent and automatically done, a similar association between scripts and their cached code fragments can be maintained.

3 Augmentation with Page Pre-Generation

As mentioned in the Introduction, caching helps to reduce page construction time after a request has been received, that is, *post-facto*. However, if it were possible to anticipate a forthcoming request and have the page generated *apriori*, then the response would be instantaneous.

There are several page access prediction models that have been proposed in the literature [11, 14, 21, 22, 24, 26]. These models can be classified into two categories: *point-based* and *path-based*. The point-based models predict the user's next request solely based on the current request being served for the user. On the other hand, the path-based prediction models are built on entire request paths followed by users. The path-based predictions use a path profile, which is a set of pairs, each of which contains a path and the number of times that path occurs over the period of the profile. The profiles can be generated from standard HTTP server logs and the accuracy of these models has been found to be high enough to justify the pre-generation of dynamic content [21] – in the rest of this paper, we assume the use of such a path-based prediction model.

Specifically, for an outgoing user response at the web-server, the web-server decides to generate the most expected next page for the user, based on many considerations such as current system load, the type of user, the benefit of pre-generating a page and so on. When the web server receives the next page request from the same user, it checks whether it has pre-generated the page the user is requesting now. If so, the page is served immediately. If not, the page request is treated as a normal page request and the page is constructed freshly and served.

The key behind the success of this approach is the delay between user requests. The user response leaving the web-server will take some time to reach the user and the user will take some time to click the next page. It is expected that this time delay is sufficient for the dynamic page pre-generation, before the arrival of the next request of the same user. In case of a correct prediction, the server latency in terms of page construction time is reduced to zero. Thus, the dynamic page pre-generation achieves the immediate benefit for the current user.

3.1 Server Load Management

While page pre-generation is useful for reducing response times, it also involves expense of computational resources. This is acceptable under normal operating conditions, even if the page prediction accuracy is not good, since web-servers are typically over-provisioned in order to be able to handle peak load conditions [21], and we are only using this excess capacity. But, when the system is under peak load conditions, the wasted resources due to the mistakes made by the pre-generation process may actually exacerbate the situation, driving the system into *a worse condition*. To address this issue, we implemented in [23] a simple linear feedback mechanism that modulates the pre-generation process to suit the current loading condition. We use this technique in the work reported here also.

Specifically, the system load is periodically measured, and if it exceeds a threshold value, the role of the page pre-generator is restricted in proportion to the excess load. For each outgoing page response, the web server allows the page pre-generator to generate pages with probability $prob\_gen$ set as follows:

$$prob\_gen = 1 \text{ if } (current\_load < threshold\_load)$$
$$prob\_gen = \frac{100 - current\_load}{100 - threshold\_load} \quad \text{otherwise}$$

When the pre-generator is restricted in the above manner, its assigned *cache partition* may become underutilized – in this case the size of the fragment-cache is dynamically enlarged to cover the underutilization of the page cache.

4 Simulation Model

To evaluate the performance of the proposed integrated-caching system, we have developed a detailed discrete-event simulator of a web-server supplying dynamic web-pages to users – this model is similar to that used in our previous work [23], and is extended to incorporate the code-caching feature. Table 1 gives the default values of the parameters used in our simulator – these values are chosen to be indicative of typical current web-sites, with some degree of scaling to ensure manageable simulation run-times.

4.1 Web-site Model

A directed graph is used to model the web-site. Each node in the graph represents a dynamic web-page, and each edge represents a link from one page to another page within the web-site. A node may be connected to a number of other nodes. The web-site graph is generated in the following manner: We start with a node called the root node, at level zero, and an initial fanout $FanOut$. Then, at each level l, for all nodes of that level, the next level nodes are created and linked, with a uniform random fanout ranging between $(0, FanOut - l)$. When a fanout of 0 is chosen at a node, the generation process at that node is terminated. In order to model "back-links", we permit, in the process of linking a node to other nodes, even the *previously generated nodes* of the prior levels to be candidates. The percentage of back links is determined by the $BackLinks$ parameter.

4.2 Web-page Model

Each dynamic web-page consists of a static part and a collection of identifiable dynamic fragments. A fraction $FragCacheable$ of these dynamic fragments are cacheable, while the remaining are not. The number of fragments in a page are uniformly distributed over the range *(MinFragNum,MaxFragNum)*. Two distributions of the choice of fragments are considered: *Uniform*, where the fragments are selected uniformly from the *FragPopulation* fragments, and *Skewed*, where a "90-10" rule applies in that 90 percent of the fragment choices are made from 10 percent of the *FragPopulation* fragments. Finally, the cost of producing a fragment, $FragCost$, is taken to be proportional to its size which is uniformly distributed over the range *(MinFragSize,MaxFragSize)*.

The fragment source-code sizes are between $MinimumSourceSize$ and $MaximumSourceSize$. When the fragment source code is compiled, the size of the compiled code is usually larger as compared to the equivalent source code size – this increase is modeled by the factor $CodeBlowupFactor$, and the value chosen is based on our analysis of a representative set of real-world scripts. The minimum execution speedup due to pre-compilation is given by $ReductionFactor$ – the value chosen is based on a conservative estimate of the speedups mentioned in [16]. We have used conservative parameter values based on the analysis on real-world scripts reported in [7, 20].

Finally, the accuracy of page access prediction, used in the page pre-generation process, is determined by the $PagePredict$ parameter. The load-controlling feedback mechanism kicks in when the current load exceeds the setting of the $ThresholdLoad$ parameter.

4.3 User Model

The web-site receives requests from the sessions of different users. The creation of sessions is assumed to be Poisson distributed [1] with rate *ArrivalRate*. Each session generates one or more page requests, in a sequential manner. The number of pages in a session are uniformly distributed over the range *(MinSessionPage, MaxSessionPage)*. The web-site receives requests from the sessions of different users. Between the page requests of a session, a uniformly distributed user think time over the range *(MinThinkTime, MaxThinkTime)* is modeled.

4.4 Cache Model

The web-server has a cache for dynamic page construction, of size $CacheSize$. The fraction of the cache given to the code-cache is given by *CodeCacheFraction*, with the remainder assigned to the fragment-cum-page cache. Within the fragment-cum-page cache, the space is equally divided between the fragments and pages, as per the recommendation in [23]. The search times in the code-cache and fragment-cum-page cache are determined by the *CacheSearchTime* parameter. The fragments in the fragment-cache are modeled to be invalidated randomly by the data source with an invalidation rate set by $InvalidRate$.

5 Experiments and Results

Using the above simulation model, we conducted a variety of experiments, the highlights of which are described here. The performance metric used in all our experiments is the average *dynamic page construction time*, evaluated for a range of fragment cacheabilities and page prediction accuracies, as a function of the session arrival rate and the fraction of the cache assigned to the code-cache. The fragment cacheability and page prediction accuracy are evaluated for the following settings: LOW (20%), MEDIUM (50%) and HIGH (80%), covering the spectrum of real-life website environments. Also, the user arrival rates cover both normal loading conditions as well as peak load scenarios.

To put the performance of our new **Integrated** approach in proper perspective, we compare it against two benchmark algorithms: **Pure_FC**, which implements pure fragment-caching on the entire cache, and **Pure_CC**, which implements pure code-caching on the entire cache, respectively.

Table 1: Simulation Parameter Settings

Parameter	**Setting**
FanOut	10
BackLinks	20 percent
FragPopulation	8000
CacheSize	2 MB
CodeCacheFraction	0 to 100 percent
FragCost	20 ms
MinPageSize	10 KB
MaxPageSize	30 KB
MinFragNum	1
MaxFragNum	19
MinFragSize	1 KB
MaxFragSize	3 KB
ArrivalRate	0 to 12 sessions per second
InvalidRate	1/ms
MinSessionPage	1
MaxSessionPage	19
MinThinkTime	1 second
MaxThinkTime	9 seconds
FragCacheable	20, 50, 80 percent
CacheSearchTime	0.1 ms
NumberofPagesUsed	2074
MinimumSourceSize	100 bytes
MaximumSourceSize	300 bytes
CodeBlowupFactor	10
ReductionFactor	2
ThreshholdLoad	75 percent
PagePredict	20, 50, 80 percent

In the initial set of experiments, page pre-generation is not included, but is considered subsequently.

5.1 Expt. 1: Page Construction Times (Uniform)

In our first experiment, we evaluate the dynamic web-page construction times for an environment where the fragment-cacheability is Medium (50 percent), the cache memory is *equally* partitioned between the code-cache and the fragment-cache (there is no page-cache), and the fragment distribution is *Uniform*.

For this scenario, Figure 2 gives the relative performance of the various algorithms as a function of the session arrival rate. We see here first that the Integrated approach performs about 20% better than Pure_CC, and 30% better than Pure_FC. Further, it can sustain *performance stability* for a higher arrival rate (upto 6 sessions per second) as compared to both Pure_CC and Pure_FC.

5.2 Expt. 2: Page Construction Times (Skewed)

When the above experiment is carried out with a *Skewed* fragment distribution, the resulting performance is as shown in Figure 3. We see here that the performance

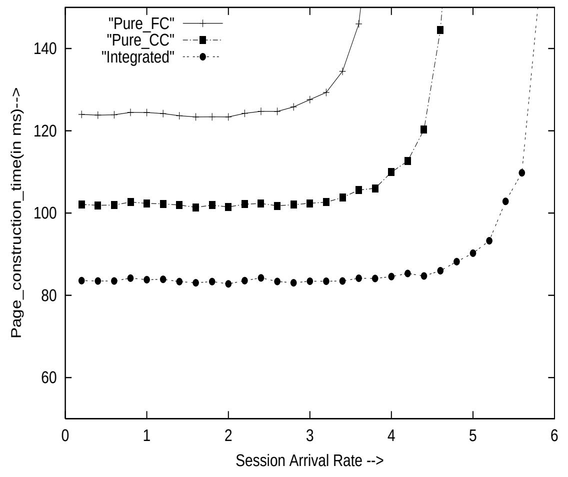


Figure 2: Page Construction Times (Uniform)

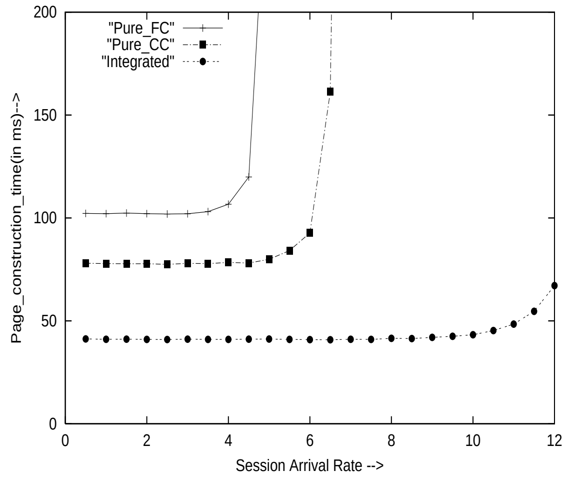


Figure 3: Page Construction Times (Skewed)

differences between Integrated and the baselines *substantially increase* – now, Integrated is 40 percent better than Pure_CC and 60 percent better compared to Pure_FC. In most real-world situations, fragment choices are indeed skewed, highlighting the importance of the Integrated approach to page construction.

Further, the Integrated algorithm sustains *performance stability* for much higher arrival rates (upto 12 sessions per second) as compared to both Pure_CC (6 sessions per second) and Pure_FC (4 sessions per second).

5.3 Expt. 3: Cache Partitioning (Uniform)

In our next experiment, we investigated the performance impact on the Integrated approach of various choices of cache partitioning sizes between the code-cache and the fragment-cache. This is done over the entire range of fragment-cacheability levels (Low, Medium and High), resulting in three different cases. The results for all these cases are shown in Figures 4(a-c) for arrival rates 1 through 4, as a function of the code-cache percentage size, and for a Uniform fragment distribution.

In these figures, we first observe that all the Integrated graphs have a "cup shape" with the highest construction times being at the extremes (0 percent code-cache and 100 percent code-cache), and the lowest somewhere in between. Further, we find that a simple heuristic relationship exists between the best partition, which leads to lowest page construction time, and the fragment-cacheability:

$$BestPartition = 100 - Fragment\_cacheability \quad (1)$$

So, for example, with High (80%) fragment-cacheability, the Best Partition occurs at about 20% for the code-cache. In all our other experiments also, we found this heuristic to approximately hold true.

An important point to note here is that the setting of 0 percent code-cache is equivalent to a *Pure_FC* approach, while 100 percent code-cache is equivalent to *Pure_CC*. We observe in the graphs that the performance of both Pure_FC and *Pure_CC* are very highly variable with regard to the fragment-cacheability level.

5.4 Expt. 4: Cache Partitioning (Skewed)

When the above experiment was carried out with a *Skewed* fragment distribution, the resulting performance is as shown in Figures 5(a-c). We see here that the cup shapes are much deeper as compared to the Uniform case, indicating that choosing the Best Partition appropriately becomes even more critical. But this is not difficult since the choice continues to be in accordance with Equation 1.

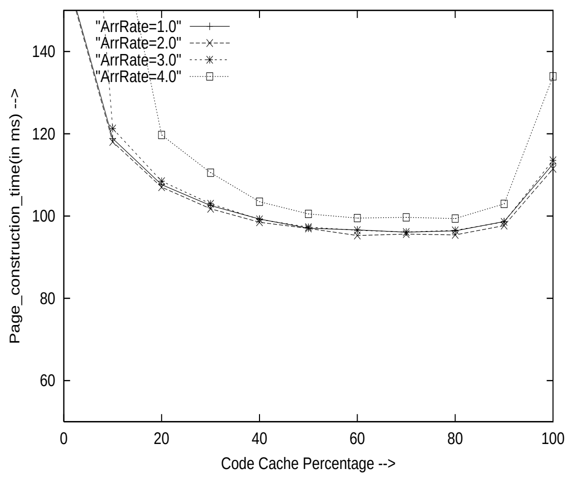
5.5 Expt. 5: Page Pre-generation (Uniform)

We now move on to investigating the impact of the optional page pre-generation on dynamic page construction times. In our first experiment here, the caching algorithms are augmented with page pre-generation for 50 percent page predictability, the rest of the parameters remaining the same as that of Expt 1 (as mentioned earlier, the internal partitioning of the fragment-cum-page cache is always *equally* split between fragments and pages, as per [23]).

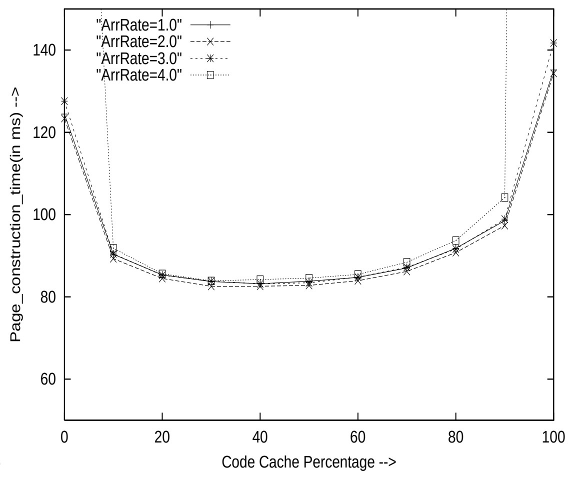
For this environment, the page construction times are shown in Figure 6 – for graph readability, we only show the performance of the basic Integrated algorithm and its PG (page pre-generation) variation. We first see here that PG_Integrated performs upto *30 percent better* than basic Integrated during normal loading (upto 2 user sessions per second), and no worse during peak loading (by virtue of the feedback-based load-control mechanism). In a nutshell, PG_Integrated provides both excellent average-case performance and stable worst-case performance.

5.6 Expt 6: Page Pre-generation (Skewed)

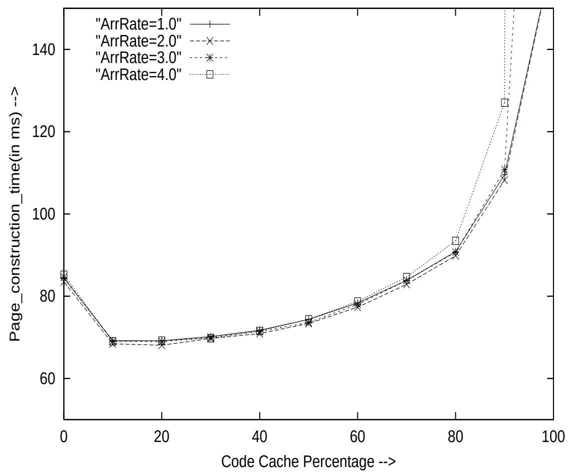
When the above experiment was carried out with a *Skewed* fragment distribution, the resulting performance is as



(a)

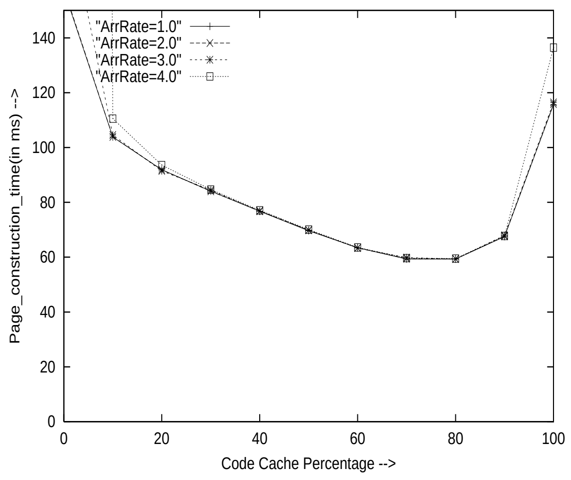


(b)

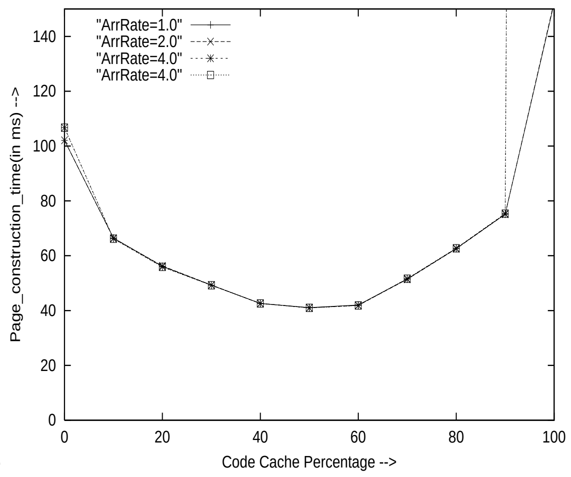


(c)

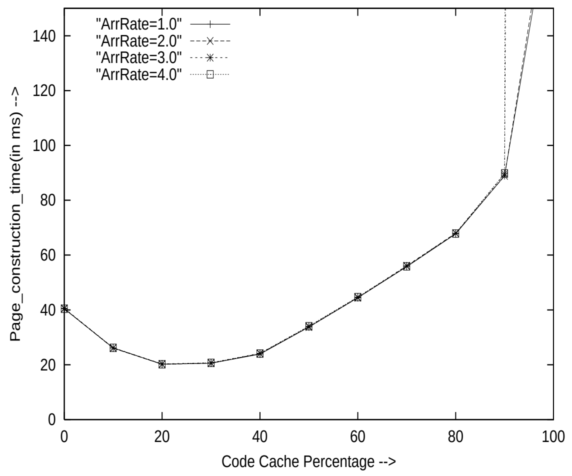
Figure 4: Cache Partitioning (Uniform)
(a) LOW fragment-cacheability (b) MEDIUM fragment-cacheability (c) HIGH fragment-cacheability



(a)



(b)



(c)

Figure 5: Cache Partitioning (Skewed)
(a) LOW fragment-cacheability (b) MEDIUM fragment-cacheability (c) HIGH fragment-cacheability

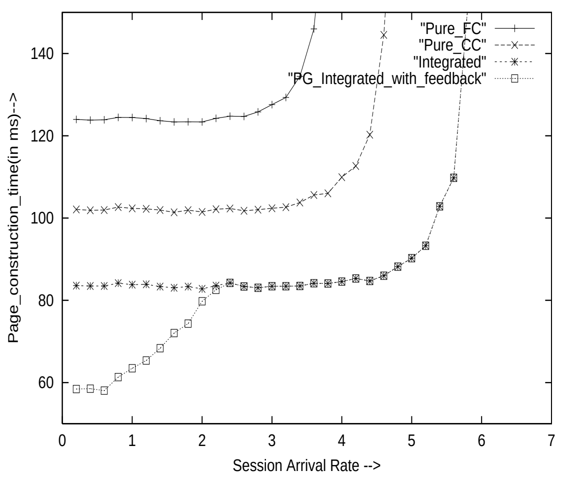


Figure 6: Impact of Page Pre-Generation (Uniform)

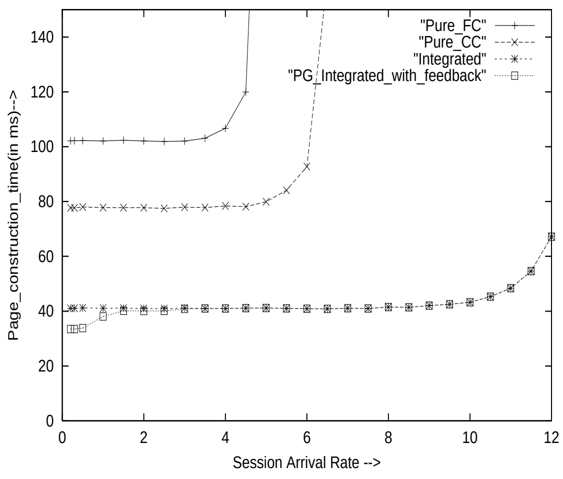


Figure 7: Impact of Page Pre-generation (Skewed)

shown in Figure 7. We see here that the performance is improved only marginally (by about 10 percent at the low loads). This is because when the fragment distribution is skewed, then most of the frequent fragments are either served from the fragment cache or executed directly from the code cache and they are the ones which are frequently requested. So there is little work remaining, even when the page is constructed after receiving the request. Hence the impact of the page pre-generator is marginal in this case.

5.7 Summary

As described above, we have carried out a variety of experiments on the proposed Integrated caching both with and without page pre-generation. To summarize our experimental results, we observe that Integrated caching (without page pre-generation) performs 20 percent better than Pure_CC(the code caching) and 30 percent better than Pure_FC(the fragment caching), under Uniform fragment distribution. Under Skewed fragment distribution, it performs 40 percent better than Pure_CC and 60 percent better than Pure_FC.

The extended Integrated caching with feedback-controlled page pre-generation performs better than the Integrated caching during normal loading and dose no worse during peak loading. We observe that during normal loading speedups of 10 to 30 percent can be realized, depending on the fragment distribution.

6 Related Work

In this section, we survey some of the literature related to the issues presented in this paper.

A widely used existing approach to address Web performance problems is based on the notion of content-caching, and a variety of such methods are discussed in [15]. Various types of database caching have been suggested, including caching database tables in main memory [30] and caching the results of database queries [17]. Database caching approaches can reduce only some of the delays associated with query processing operations. In [25], a caching system that caches content at various levels, such as database queries, HTML fragments, and pages is proposed. This work concentrates primarily on the declarative specification of websites and offers a number of tools for the easy implementation, deployment and monitoring of these sites. Another approach is presentation layer caching, which caches HTML fragments. Many application servers provide this type of caching capability (e.g., WebLogic from BEA Systems [28]), which can mitigate delays due to presentation layer tasks. An efficient technique to compose web pages from fragments for web-based publications is given in [4] and this scheme was implemented in the 2000 Olympic Games web-site hosted by IBM. Fragment-based web publication allows parts of dynamic web-pages to be cached. All of the above-mentioned caching approaches can help to reduce the delays associated with content generation. Since they reside at the origin server, these solutions do guarantee the correctness of the content.

There are two broad approaches that use proxies to cache dynamic pages, namely page-level caching and dynamic page assembly. In page-level caching, the proxy caches the complete page outputs of dynamic sites. Page-level caching has been considered in [2, 3, 32]. The page-level caches can improve web-site performance by reducing delays associated with page generation. However, there are some major limitations associated with using page-level caching. The most important limitation of proxy-based page level caching is serving incorrect pages, whenever the page level caching must rely on the request URL to identify pages in cache. When pages are dynamically generated, different invocations of a given script are not guaranteed to produce the same page [8]. Another limitation of page level caching solutions is that there is often very little reusability of full HTML pages. Specially, the sites that serve highly personalized pages may make every page instance unique and reusable only if the same user makes the same request.

This can lead to low hit ratios. Page level caching may also lead to unnecessary invalidation. If only one or a few elements on a page become invalid, then the entire page becomes invalid [8].

Dynamic page assembly is an approach popularized by Akamai [27] as part of the Edge Side Includes (ESI) initiative [29]. This approach entails establishing a template for each dynamically generated page. The template specifies the content and layout of the page using a set of markup tags. A drawback of this approach is the requirement that a site follow a specified page design paradigm, specifically, the use of templates which in turn call separate dynamic scripts for each dynamically generated fragment, forcing that page layout be known in advance. Thus, sites supporting dynamic layouts will not be able to take the advantage of dynamic page assembly. Another drawback of the dynamic page assembly approach is that it cannot be used in the context of pages with semantically interdependent fragments. The approach presented in [18] can be considered to be a dynamic page assembly approach. This work proposes a proxy cache that stores query templates, along with query results, which are used to manage the cache. This approach can only mitigate some delays associated with query processing, but it does not address the other delays associated with dynamic page generation.

Overall, the proxy caching approaches can provide significant bandwidth savings, but their applicability in caching dynamic pages is rather limited and their primary use is in caching fixed layout content.

There are several page access prediction models that have been proposed in the literature, based on information gained from mining web logs. Several Markov models derived from the behaviour patterns of many users, which predict which documents a user is likely to request next, are presented in [26]. Based on an evaluation of their predictive accuracy, hybrid models which combine the individual models in different ways are derived and shown to have greater predictive accuracy. The use of path profiles for describing HTTP request behavior has been introduced along with an algorithm for efficiently creating these profiles, in [21]. This paper also claims that predictive accuracies are high enough to justify generating dynamic content before the client requests it. An n-gram based model to develop path profiles of users from very large data sets is presented in [22].

Web prefetching is an alternative technique to web caching used to reduce the noticeable response time perceived by users. Prefetching is often proposed in an attempt to retrieve objects in advance of a client request. This idea has been implemented in a number of browser add-ons. The prefetch techniques in various contexts are discussed in [11, 14, 24]. The prefetching implementations can cause problems with undesirable side-effects and server abuse. Any unused prefetched page increases both bandwidth consumption and load on the server. The prefetching technique, if used for enriching the client side cache,may be useful in case of static contents. Client side caching is not at all useful for dynamic web pages. For the dynamic web pages cached at client side, there is no way for the server to invalidate, whenever the page/page-fragments become invalid at the origin server.

In our earlier work [23], we have studied the efficient combination of fragment-caching with page pregeneration. Our experiments have shown that significant page response time reduction is possible by splitting the cache evenly between the page and fragment partitions.

Code-caching is a recent technique proposed in the context of script execution. Before executing a script, the script execution engine reads, parses, and compiles the contents into compiled code ready for execution. Since, in practice, the scripts rarely change, this pre-processing is targeted for elimination in code caching. Such techniques have been recently implemented in the context of the PHP Accelerator project [16], JIT compiler of JVM [33], etc. In these cases it is assumed that the compiled code uses instructions from a virtual instruction set that is platform-independent and once compiled, the code is executed by a virtual machine that interprets the instructions. The code-caching technique used by ScriptBasic [31] stores object code for scripts written in BASIC.

7 Conclusions

In this paper, we have proposed a simple integrated caching approach to reduce dynamic web-page construction times by appropriately combining both fragment-caching and code-caching. We made a detailed evaluation of the integrated caching approach over a range of cacheability levels and fragment choice distributions. Our experimental results showed that the integrated approach can provide significant reductions in construction times, especially for skewed fragment distributions. We were also able to identify a simple heuristic for identifying the appropriate size of the code cache partition.

We extended our integrated caching model to incorporate anticipatory page pre-generation with feedback-based load control. The results of this enhancement indicate that the extended approach performs significantly better during normal loading and does no worse during peak loading.

In summary, the techniques proposed in this paper achieve long-term benefits through fragment and code-caching and immediate benefits through page pregeneration. Currently, our work uses a LRU-based cache replacement policy. If web-logs for fragment access become available, then it will be interesting to study the performance of customized cache-replacement policies. Further, our current results are based on simulation experiments. In our future work, we plan to obtain a more realistic assessment of the performance improvements by deploying the proposed integrated policy in an actual web server.

Acknowledgements

This work was supported in part by a Swarnajayanti Fellowship from the Dept. of Science & Technology, Govt. of India. We

thank the anonymous reviewers for their constructive comments.

References

[1] M. Andersson, J. Cao, M. Kihl and C. Nyberg, "Performance Modeling of an Apache Web Server with Bursty Arrival Traffic", *Proc. of Intl. Conf. on Internet Computing, June 2003.*

[2] K. Candan, W. Li, Q. Luo, W. Hsiung and D. Agrawal, "Enabling dynamic content caching for database-driven web sites", *Proc. of ACM SIGMOD Conf., May 2001.*

[3] J. Challenger, P. Dantzig and A. Iyengar, "A scalable system for consistently caching dynamic web data", *Proc. of 18th Annual Joint Conf. of the IEEE Computer and Communications Societies, March 1999.*

[4] J. Challenger, A. Iyengar, K. Witting, C. Ferstat and P. Reed, "A Publishing System for Efficiently Creating Dynamic Web Content", *Proc. of IEEE INFOCOM Conf., March 2000.*

[5] Chutney Technologies, Inc. "Dynamic Content Acceleration: A Caching Solution to Enable Scalable Dynamic Web Page Generation", *Proc. of ACM SIGMOD Conf., May 2001.*

[6] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, K. Ramamritham and D. Fishman, "A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration", *Proc. of 27th VLDB Conf., September 2001.*

[7] A. Datta, K. Dutta, H. Thomas, D. VanderMeer and K. Ramamritham, "Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation", *ACM Trans. on Database Systems, Vol. 29, No. 2, June 2004.*

[8] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham, "Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation", *Proc. of ACM SIGMOD Conf., June 2002.*

[9] A. Eden, B. Joh and T. Mudge, "Web Latency Reduction via Client-Side Prefetching", *Proc. of IEEE Intl. Symp. on Performance Analysis of Systems and Software, April 2000.*

[10] Ejacent White paper, "Resilient Performance and Instant Scalability for Interactive Web Site Services: A New Approach to Internet Infrastructure for Dynamic Content and Services", *http://www.ejacent.com.*

[11] D. Duchamp, "Prefetching Hyperlinks", *Proc. of 2nd USENIX Symp. on Internet Technologies and Systems, October 1999.*

[12] C. Huitema, "Network vs. server issues in end-to-end performance", *Keynote address, Workshop on Performance and Architecture of Web Servers, June 2000.*

[13] A. Iyengar and J. Challenger, "Improving Web Server Performance by Caching Dynamic Data", *Proc. of Usenix Symp. on Internet Technologies and Systems, December 1997.*

[14] Z. Jiang and L. Kleinrock, "Prefetching Links on the WWW", *Proc. of IEEE Intl. Conf. on Communications, June 1997.*

[15] C. Mohan, "Caching technologies for web applications", *Tutorial, 27th VLDB Conf., September 2001, http://www.almaden.ibm.com/u/mohan/Caching_VLDB2001.pdf*

[16] N. Lindridge, "The PHP Accelerator 1.2", *http://www.php-accelerator.co.uk/PHPA_Article.pdf., April 2002.*

[17] Q. Luo, J. Naughton, R. Krishnamurthy, P. Cao and Y.Li, "Active query caching for database web-servers", *Proc. of 3rd WebDB Workshop, May 2000.*

[18] Q. Luo and J. Naughton, "Form-based proxy caching for database-backed web sites", *Proc. of 27th VLDB Conf., September 2001.*

[19] V. Panteleenko and V. Freeh, "Instantaneous Offloading of Transient Web Server Load", *Proc. of 6th Intl. Workshop on Web Caching and Content Delivery, June 2001.*

[20] L. Ramaswamy, A. Iyengar, L. Liu and F. Douglis, "Automatic Detection of Fragments in Dynamically Generated Web Pages", *Proc. of 13th Intl. World Wide Web Conf., May 2004.*

[21] S. Schechter, M. Krishnan and M. Smith, "Using Path Profiles to Predict HTTP Requests", *Proc. of 7th Intl. World Wide Web Conf., April 1998.*

[22] Z. Su, Q. Yang, Y. Lu and H. Zhang, "WhatNext: A Prediction System for Web Requests using N-gram Sequence Models", *Proc. of 1st Intl. Conf. on Web Information Systems Engineering, June 2000.*

[23] Suresha and J. Haritsa, "On Reducing Dynamic Web Page Construction Times", *Proc. of 6th APWEB Conf., April 2004.*

[24] Z. Wang and J. Crowcroft, "Prefetching in World Wide Web", *Proc. of IEEE Global Telecommunications Internet Mini-Conf., November 1996.*

[25] K. Yagoub, D. Florescu, V. Issarny and P. Valduriez, "Caching Strategies for Data-Intensive Web Sites", *Proc. of 26th VLDB Conf., September 2000.*

[26] I. Zukerman, D. Albercht and A. Nicholson, "Predicting Users' Requests on WWW", *Proc. of 7th Intl. Conf. on User Modeling, June 1999.*

[27] Akamai Technologies. *http://www.akamal.com*

[28] BEA Systems, "Weblogic application server", *http://www.bea.com/products/weblogic/index.html*

[29] ESI Consortium, "Edge side includes", *http://www.esi.org*

[30] Oracle Corp, "Oracle 9ias database cache", *http://www.oracle.com/ip/deploy/ias/d_cache_fov.html*

[31] Script Basic, "Code Caching", *http://www.scriptbasic.com/html/texi/ug/ug_4.html*

[32] SpiderCache. *http://www.spidercache.com*

[33] Sun Microsystems, "Java servlets and jsp", *http://java.sun.com*