Efficiently Serving Dynamic Data at Highly Accessed Web Sites

James R. Challenger, Paul Dantzig, Arun Iyengar, *Senior Member, IEEE*, Mark S. Squillante, *Fellow, IEEE*, and Li Zhang, *Member, IEEE*

***Abstract*—We present architectures and algorithms for efficiently serving dynamic data at highly accessed Web sites together with the results of an analysis motivating our design and quantifying its performance benefits. This includes algorithms for keeping cached data consistent so that dynamic pages can be cached at the Web server and dynamic content can be served at the performance level of static content. We show that our system design is able to achieve cache hit ratios close to 100% for cached data which is almost never obsolete by more than a few seconds, if at all. Our architectures and algorithms provide more than an order of magnitude improvement in performance using an order of magnitude fewer servers over that obtained under conventional methods.**

***Index Terms*—Caching, dynamic content, performance analysis, prefetching, stochastic models, Web sites.**

I. Introduction

MANY Web sites today have an increasing need to serve dynamic content. Dynamic content is important for Web sites that provide rapidly changing information, e.g., sports Web sites must provide the latest information about sporting events, and financial Web sites must provide current information about stock prices. If the pages for such Web sites are generated dynamically by a server program that is executed every time the pages are requested, the program can return the most recent version of the dynamic content, whereas if files are created to serve the pages statically, it may not be feasible to keep the files up to date. This is particularly true if there are a large number of files that need to be frequently updated. Dynamic content is also important for creating Web pages on the fly from databases. Search engines satisfy queries dynamically from databases. Web pages corresponding to product catalogs are often created dynamically from databases. Information personalized to individual users is also frequently created dynamically.

While the benefits of dynamic pages are clear in many Web environments, dynamic pages can seriously reduce Web server performance, even if the dynamic content only comprises a fraction of all requests. Although high-performance Web servers can typically deliver up to several hundred static files per second on a uniprocessor, the rate at which dynamic pages are delivered is often orders of magnitude slower. In fact, it is not uncommon for a program to consume over a second of CPU time in order to generate a single dynamic page. For Web sites with a high proportion of dynamic pages, the performance bottleneck is often the CPU overhead associated with generating dynamic pages [1]–[3].

An important technique for improving performance at Web sites generating significant dynamic content is to cache dynamic pages the first time they are created. That way, subsequent requests for the same dynamic page can access the page from a cache instead of repeatedly invoking a program to generate the same page. A key problem with caching dynamic pages, however, is determining what pages should be cached and when a cached page has become obsolete. The optimal solution to this problem depends heavily upon the request and update patterns for each page. For example, pages that are requested frequently but updated infrequently might be good candidates for caching, whereas pages that are updated frequently but requested infrequently might be poor candidates for caching. Moreover, an update to some pages might be an indicator of a burst of requests for the pages in the very near future, and intelligent cache prefetching can be an important component in improving performance. On the other hand, a page that is more likely to be updated before it is next referenced might be a good candidate for cache replacement.

One of the problems encountered at production Web sites concerns maintaining good system performance after new information is received, as it can often be difficult to precisely identify which cached pages are changed due to the new information. In order to insure that all stale pages are invalidated, many up-to-date pages may also be invalidated. This can cause high miss ratios after the system receives new information [1].

To solve these problems, we have developed *Data Update Propagation (DUP)* for precisely identifying the set of cached pages that have become obsolete as a result of new information received by the system, together with a collection of related algorithms for cache management that includes page prefetching and invalidation. Our DUP algorithms significantly reduce the number of cached pages that need to be invalidated or updated after new information is received. DUP is a critical component of general multitier architectures deployed for high-performance serving of dynamic content to many clients at several highly accessed Web sites (HAWS) hosted by IBM. Whenever new content becomes available, updated Web pages reflecting these changes are made available to the rest of the world within a few seconds, which can be adjusted to meet the needs of the application environment. Clients can thus rely on the Web site to provide the latest results, news, photographs,

Manuscript received November 1, 2001; revised January 16, 2003; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Crovella.

The authors are with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: challngr@us.ibm.com; PaulDantzig@us.ibm.com; aruni@us.ibm.com; mss@watson.ibm.com; zhangli@us.ibm.com).

Digital Object Identifier 10.1109/TNET.2004.826289

and other information. The degree of consistency we achieve is sufficient for a large variety of Web sites providing information such as news stories, stock quotes, and sports results. High availability is achieved via redundant hardware and intelligent routing techniques.

The design of our system architectures and algorithms is based on a detailed analysis of the stochastic properties of the client request patterns and the dynamic page update patterns at a few previous HAWS providing significant dynamic content, including the 1998 Nagano Olympic Games Web site [4] (henceforth, *Nagano*). Such content dynamics are particularly important because of their deep implications on the effectiveness of Web caching mechanisms [5]. In addition to motivating system design decisions, our analysis is exploited in the on-line algorithms to dynamically adjust real-time policy decisions for caching, prefetching and invalidation. We also have experience with these architectures and algorithms for caching dynamic content at many subsequent HAWS, including those from the 2000 Sydney Olympic Games [6] (henceforth, *Sydney*) up to the most recent Web sites for the U.S. Open Tennis, Wimbledon, Australian Open, French Open, Master's Golf, Ryder Cup, Tonys, and Grammys. The system design also has been successfully deployed at various financial Web sites.

Our experience with this system design has proven to be consistent with the results of our stochastic analysis, and thus we present a representative sample of some of these results and discuss their impact on our design. In addition to exploiting these models and results in our system design, the results of our analysis are of interest in their own right especially given the limited previous research in the literature on update patterns, as well as their interactions with the corresponding request patterns, in highly dynamic Web sites. Specifically, our results show that the stochastic properties of the client request patterns and dynamic update patterns can differ significantly from one page to another and vary over time. This includes a wide range of tail distributions for the interrequest and interupdate times, with some cases exhibiting heavy-tail distributions, as well as seasonal and strong dependence structures that can represent the burstiness and correlation of such request and update patterns. We further show that there tend to be correlations between the updates for certain pages and subsequent requests for the page immediately thereafter, which are exploited in our prefetching and caching algorithms. We therefore track, model, and exploit this information throughout our system for efficiently serving dynamic content.

We also present another aspect of our quantitative analysis upon which the design of our architectures and algorithms is based. In particular, we investigate the performance benefits of our system design for efficiently serving dynamic content at a representative previous HAWS, thus demonstrating the significant improvements provided by our approach. A detailed simulation model is used together with traces from the Nagano Web site [4] and measurement data. The results of our analysis show that the architectures and algorithms presented in this paper provide more than an order of magnitude improvement in performance using an order of magnitude fewer servers over that realized under conventional methods for serving dynamic content. By exploiting the DUP algorithms to maintain up-to-date copies of the dynamic pages cached at each server, the system can serve dynamic content at the performance level of serving static content. Moreover, by exploiting the DUP prefetching algorithms, we are able to achieve cache hit ratios close to 100%. Such high cache hit ratios allow Web sites to serve pages quickly even during peak request periods. In addition to being validated against the performance exhibited at the Nagano Web site, the results of our performance analysis is consistent with the corresponding results from subsequent deployment of the system design at more recent HAWS.

The remainder of the paper is structured as follows. Sections II and III present our DUP algorithms and our system architectures, respectively. Section IV presents our stochastic analysis of the request and update patterns, followed by a performance analysis of our system design presented in Section V. Section VI discusses some related work. Section VII briefly summarizes our main results and conclusions.

II. Reducing Dynamic Data Overhead

A. Caching

Caching has been successfully deployed to improve Web performance for static content, but dynamic objects are harder to cache because they change frequently. The DUP algorithms are developed to solve problems related to determining what dynamic pages should be cached and when a cached page has become obsolete. Our analysis shows that, despite the higher update rates for dynamic content, the number of requests for popular dynamic pages far exceed the number of updates to those pages. Hence, judicious use of caching can significantly reduce the number of times such pages have to be regenerated by a server.

DUP determines how cached Web pages are affected by changes to underlying data that determine the current contents of the pages. For example, a set of several cached Web pages may be constructed from tables belonging to a database. In this situation, a method is needed to determine which Web pages are affected by updates to the database. That way, caches can be synchronized with databases so that they do not contain stale data. Furthermore, the method should associate cached pages with parts of the database in as precise a fashion as possible. Otherwise, objects whose contents have not changed may be mistakenly invalidated or updated from a cache after a database change. Such unnecessary updates to caches can increase miss ratios and degrade performance.

DUP maintains correspondences between *objects* that are defined as items which may be cached and *underlying data* that periodically change and affect the contents of objects. Objects typically include Web pages or parts of Web pages, but may be other entities as well. A dynamically generated Web page created by a server program is what would typically be stored in a cache. The cache may also contain a fragment of a dynamically generated Web page which is assembled to create a complete page. If Web pages are being constructed from databases, underlying data would typically include database tables. If complex Web pages are being constructed from simpler fragments [7], the fragments would constitute underlying data.

The system maintains data dependence information between objects and underlying data. When the system becomes aware of a change to underlying data, it queries the dependence information that it has stored in order to determine which cached objects are affected. Caches use this dependency information, together with the current update and request characteristics for each of these objects, to determine which objects need to be invalidated or updated as a result of changes to underlying data.

Information for managing cached objects is maintained by a *cache manager* which is typically implemented as a long-running daemon process. Application programs communicate with cache managers in order to add or delete items from caches. Application programs are also responsible for communicating data dependencies between underlying data and objects to cache managers. Such dependencies can be represented by a directed graph known as an *object dependence graph (ODG),* wherein a vertex usually represents an object or underlying data. An edge from a vertex v to another vertex u, denoted (v, u), indicates that a change to v also affects u. Node v is known as the *source* of the edge, while u is known as the *target* of the edge. For example, if node $go2$ in Fig. 1 changes, then nodes $go5$ and $go6$ also change. By transitivity, $go7$ also changes.

Application programs are also responsible for communicating other data used to maintain information about each cacheable object. For example, edges may optionally have weights associated with them which indicate the importance of data dependencies. In Fig. 1, the data dependence from $go1$ to $go5$ is more important than the data dependence from $go2$ to $go5$ because the former edge has a weight that is five times the weight of the latter edge. Weights are used to quantitatively determine how obsolete an object is. For deployments of DUP not requiring this, the weights are not needed.

Additional information beyond weights can be maintained by the caches to direct policy decisions in a dynamic manner. As a particularly important example, simple models of the current request and update characteristics can be maintained for each cacheable object. These models are then used to determine whether such an object is updated or simply invalidated when there are changes to the underlying data. These models can also be used to set other parameter values associated with an object. As the request and update patterns for each cacheable object change over time, the corresponding models are adjusted to reflect these changes which in turn direct policy decisions for each object in a dynamic manner.

In several of the cases we have encountered, the ODG is *simple* which means that edges do not have weights and all paths are of length one. For these situations, a simpler form of DUP described in [8] can be used.

The generalized DUP algorithms are used when the ODG is not simple and has a number of useful features, including the following.

- In some cases, it is acceptable for cached objects to be slightly out of date. For example, several minor updates or corrections might have been made to a series of Web pages. When enough changes have been made, only the new versions should be visible. For a small number of changes, however, retaining the previous objects in a cache can be significantly cheaper than always updating or invalidating the cache after every minor change. At some point, an object will become highly obsolete and will have to be invalidated or updated in the cache. A quantitative method is needed for determining when a cached object has become highly obsolete. The generalized DUP algorithms provide such a method that is based on a metric for determining how obsolete a cached object is together with simple models of the request and update characteristics for the object. In order to enable the metric, edges of the ODG should have weights that are correlated with the importance of dependencies such as in Fig. 1.
- A cache manager might be managing multiple caches. In this situation, the generalized DUP algorithms allow a single ODG to be applied to multiple caches. Different caches may apply different criteria for determining when an object has become highly obsolete. Therefore, different caches may concurrently be storing different versions of the same object.

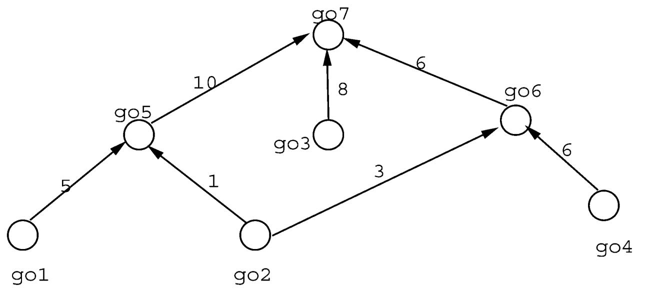

Fig. 1. Object dependence graph. Weights are correlated with the importance of data dependencies.

When an application program notifies a cache manager that underlying data has changed, the cache manager identifies all objects that are affected by finding all nodes N reachable from the nodes corresponding to the underlying data which has changed. N can be found using graph traversal techniques similar to depth-first search or breadth-first search.

If the application requires an object to be deleted from a cache whenever its value changes in any way, then identifying N is sufficient. It is not necessary to make use of weights. Otherwise, the degree to which a version $o1_1$ of an object $o1$ is obsolete is determined from the sum of the weights of edges terminating in $o1$ from nodes $n2$ for which $o1_1$ is consistent with the latest version of $n2$. If this sum falls below a threshold value, $o1_1$ is highly obsolete and should be invalidated or replaced with a more recent version. Under our definition, version $v1$ of object $o1$ and $v2$ of object $o2$ are consistent with each other if either: i) both $v1$ and $v2$ are current; or ii) at some point in the past, both versions were current.

1) Data Maintained by the Generalized DUP Algorithms: The cache manager maintains a single global directory for all caches it is managing. Each cache also maintains a local directory. The global directory maintains the following fields for objects:

- *current_version_num*: An integer representing the current version of the object. When an object is updated, the version number of the new object is the version number of the previous version incremented by 1.

- *timestamp*: Represents the time of the last change to the object. While clocks can be used for determining timestamps, the preferred method for determining the current timestamp is the number of times an application program has notified the cache manager of changes to underlying data.

Global directory entries may also exist for nodes in an ODG G which do not correspond to objects since by our definition, objects are items which may be cached.

A cache's local directory maintains the following fields for each cached object $o1$:

- *version_num*: An integer representing the version number of the object.
- *actual_sum_weight*: The sum of the weights of all edges to $o1$ from a node $n2$ such that the cached version of $o1$ is consistent with the current version of $n2$.
- *threshold_weight*: This quantity is used to determine if a version $o1_1$ of the object is highly obsolete. Version $o1_1$ is highly obsolete if the sum of the weights of edges terminating in the object from nodes $n2$, such that $o1_1$ is current with respect to $n2$, falls below *threshold_weight*. An object is current with respect to a node in G if the object is current or if no updates have been made to the node since the object became noncurrent. Highly obsolete versions should be invalidated or replaced with a more recent version. The proper value for these parameters is dependent on various trade-offs among the desired degree of consistency, the available computational resources for regenerating objects, and the request and update characteristics for the involved objects. If the system has limited computational resources and it is not a problem for Web pages to be somewhat outdated, one would tend to use low values for *threshold_weight*. On the other hand, if the system has sufficient computational resources, then higher values of *threshold_weight* will result in greater consistency, and this can be tailored to the application environment of interest. Finally, different caches may specify different values for these parameters.
- For each edge terminating in the node corresponding to $o1$ from a node $n2$, *consistent* is a Boolean field indicating whether the cached version of $o1$ is consistent with the current version of $n2$.

2) Propagating Changes to Underlying Data: Whenever underlying data changes, the application program informs the cache manager of the changes via an API function call. Let *changed_node_list* be a list of all nodes in G corresponding to underlying data which has changed. The cache manager must traverse all edges reachable from a node in *changed_node_list* in order to correctly propagate changes to all cached objects.

The cache manager maintains a counter *num_updates* for the number of updates which it is aware of. This counter is incremented whenever the cache manager is informed of new updates to underlying data. In response to such an update, all nodes on *changed_node_list* are visited first. For each such node $n1$, the cache manager increments the *current_version_num* field in the global directory by 1. The *timestamp* field in the global directory is set to *num_updates*. This indicates that $n1$ has been visited during the current graph traversal. If $n1$ corresponds to an object, the cache manager notifies all caches containing the object that the object has changed. Each cache containing the object can then invalidate its version or obtain a more recent version of the object. After all nodes on *changed_node_list* have been visited, the cache manager must traverse all edges reachable from these nodes. It can do so using graph traversal techniques such as depth-first search or breadth-first search [9].

For each edge $(n1, n2)$ which is traversed, the cache manager determines if $n2$ has already been visited. This is true if and only if the *timestamp* field in the global directory for $n2$ is equal to *num_updates*. If $n2$ has not been visited yet, its *current_version_num* field is incremented by 1, and its global directory *timestamp* field is set to *num_updates*. If $n2$ corresponds to an object $o2$, caches containing $o2$ might have to update their local directory entries for $o2$ and possibly update or invalidate their copies of $o2$. For each cache containing $o2$, the *actual_sum_weight* field is decremented by the weight of the edge $(n1, n2)$ if and only if the *consistent* field in the local directory corresponding to the edge is true. If this results in the *actual_sum_weight* being less than the *threshold_weight* field, $o2$ is either invalidated from the cache or replaced with a current version. This decision is made based on the request and update models being maintained for $o2$. For example, when an object has a high likelihood of many references before its next update, then one would tend to update the object rather than invalidate it. On the other hand, if the object has a high likelihood of many updates before its next reference, then one would tailor the decisions for the object to capture these effects and avoid useless regeneration of the object. If *actual_sum_weight* $\geq$ *threshold_weight,* $o2$ is not replaced with a new version. Instead, the *consistent* field in the local directory corresponding to the edge $(n1, n2)$ is set to false.

If, on the other hand, $n2$ has already been visited, a slightly different procedure is followed. If $n2$ corresponds to an object $o2$, caches containing $o2$ might still have to update their local directories for $o2$ and possibly update or invalidate their copies of $o2$. For each such cache, the cache manager determines if the cache contains a current version of o2 by comparing the *version_num* field in the local directory with the *current_version_num* field in the global directory. If the cached version of $o2$ is not current, the *actual_sum_weight* field is decremented by the weight of the edge $(n1, n2)$ if and only if the *consistency_level* field corresponding to the edge is true. If this results in the *actual_sum_weight* field being less than the *threshold_weight* field, $o2$ is either invalidated from the cache or replaced with a current version. This decision is made based on the request and update models being maintained for $o2$, as noted above. If *actual_sum_weight* $\geq$ *threshold_weight*, $o2$ is not replaced with a new version. Instead, the *consistent* field corresponding to the edge $(n1, n2)$ is set to false.

The amount of state information maintained by DUP is $O(|V| + |E|)$ where $|V|$ is the number of vertices in the ODG and $|E|$ is the number of edges. This includes compact representations of the request and update models for each vertex in the ODG. If Web objects are on the order of thousands of bytes, then the space overhead of DUP would likely be minimal compared to the overhead consumed by the objects themselves. If there are a high percentage of small objects and the number of edges between nodes is large, the space overhead becomes more significant. Our experience so far has been that the Web objects consume significantly more space than the additional state information maintained by DUP.

B. Prefetching

One of the key techniques we use to obtain high cache hit ratios is to calculate and cache new versions of frequently referenced pages, relative to their update characteristics, immediately after it is determined that the pages are obsolete instead of invalidating the pages and waiting for them to be loaded on demand. Consequently, once a frequently requested page is cached, the large number of future requests for the page always result in a cache hit.

This technique is effective because popular dynamic pages are often requested far more frequently than they are updated, as demonstrated by our analysis of request and update patterns. In particular, we show that pages for sports which were in progress changed as often as once or twice per minute, whereas requests for the "sports-in-progress" pages tended to arrive at rates of up to several thousand per second during peak periods. Furthermore, there tend to be correlations between the updates for certain pages and subsequent requests for the page immediately thereafter, which is further exploited in our prefetching and caching algorithms.

During the 1996 Summer Olympic Games [10], we simply invalidated cached pages when the data changed, relying on the demand to cause a cache miss and rebuild the page. Since most pages take 500–2000 ms to render, we would experience many cache misses in the time interval between invalidating a page and replacing it in cache. Each such miss caused the page to be rebuilt; the same page was therefore rebuilt and replaced many times for each invalidation.

In more recent HAWS, whenever data changed, we used the DUP algorithms to first identify the pages affected by the data change. The appropriate pages were regenerated and the stale pages replaced in cache in a single atomic operation. Cache misses were almost never observed; therefore, even during peak periods, the system was not particularly busy and had considerable excess capacity.

When a page changed, our system would regenerate the page once and store the updated page in multiple caches. Multiple caches were needed to satisfy the large number of requests to the site. Since pages only need to be generated once regardless of the number of caches, our system scales efficiently as more caches are needed to handle more requests. By contrast, the conventional approach of demand-based caching with caches operating autonomously would cause a new page generation each time a page is added to a cache. As the number of caches increases, the overhead from redundant page generations required to store current versions of the same object in different caches would become significant.

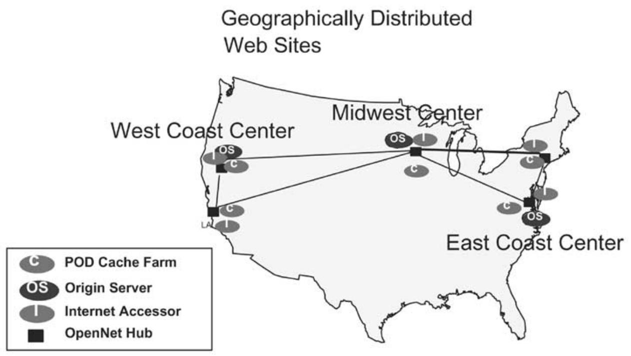

Fig. 2. Geographically distributed Web sites are used by our system.

C. Interfaces for Creating Dynamic Web Data

Interfaces for invoking server programs that create dynamic Web pages can have a significant effect on performance. The Common Gateway Interface (CGI) was once the most widely used interface for invoking server programs. It is extremely slow, however, because a new process must be created each time a server program is invoked. Instead, we use faster interfaces which incur significantly less overhead than CGI.

One lighter weight approach is for the server programs to execute as part of the Web server's process. This can be accomplished by dynamically loading server programs by the Web server or statically linking server programs as part of the Web server's executable image. Another approach is to run server programs as long-running processes with which a Web server communicates. More information about these approaches and a comparison of them is contained in [11].

III. Architectures for Serving Dynamic Data

We now describe the architectures for many of the HAWS that have served dynamic data to very large numbers of clients and have consistently been among the world's most popular Web sites. The Web sites typically utilize three to four IBM SP2 multiprocessor systems each containing a maximum of about 125 processors, over 100 GB of memory, and several terabytes of disk space. Three of these sites are permanently located in the USA; see Fig. 2. A fourth site was added in Japan (specifically, Tokyo) for Nagano [4] and in Australia for Sydney [6] to further improve performance. The SP2 multiprocessor system at each site is composed of three to four SP2 frames, each of which consists of around 10 uniprocessor nodes and 1 symmetric multiprocessor node (primarily used to handle updates as described below). In addition, since 1999 (and, in particular, after Nagano), around 100 fast reverse proxy caches [12] have been geographically distributed in five locations co-located with telecommunications providers.

This amount of hardware is deployed both for performance purposes and for high availability. Performance considerations are paramount not only because these Web sites are some of the most popular sites during busy periods but also because the data being presented to clients is constantly changing. Whenever new content is entered into the system, updated Web pages reflecting these changes are made available to the rest of the world within

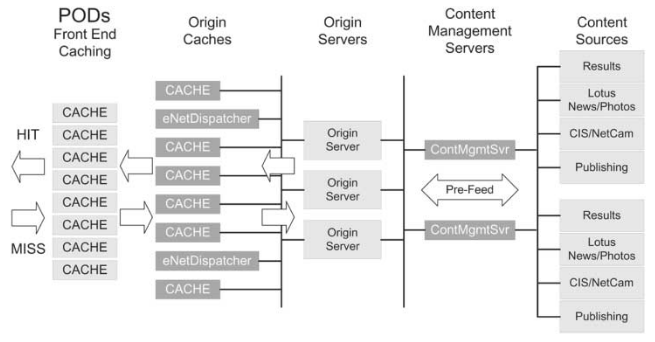

Fig. 3. Multitiered serving architectures used by our system. The two large boxes in the origin cache tier marked *eNetDispatcher* are load balancers routing requests to the origin servers.

a few seconds. This time interval tolerance can be fine-tuned to meet the needs of the application environment. Clients can thus rely on the Web site to provide the latest results, news, photographs, and other information from these events. These systems are able to serve pages to clients quickly during the entire event even during peak periods. In addition, these sites have been available 100% of the time.

The HAWS use multitier architectures, as shown in Fig. 3. The content management tier maintains cached versions of objects that are updated using the DUP algorithms. As changes come into the content management system, all pages affected by the change are updated according to our algorithms. The content distribution system then distributes all of the affected pages to the origin server layer keeping each geographic site consistent with all others to within 5 s. More recent HAWS also have two tiers of caches. Origin caches are reverse proxies in proximity to the origin servers. Point of Distribution (POD) caches are distributed throughout the network and are remote from the servers. A request first comes into a POD. In the event of a POD cache miss, the request is forwarded to an origin cache. The request only goes to an origin server in the event of both a POD cache miss and a subsequent origin cache miss. Cache hit ratios of over 90% at the POD caches are typically achieved, as demonstrated by our quantitative performance analysis. Our analysis further shows that, of the requests resulting in POD cache misses, about 50% result in origin cache hits. A detailed analysis of the caching effects for the Sydney site [6] can be found in [13].

Fig. 3 illustrates how content management and updates to underlying data are performed on different processors (specifically, heavy duty servers using multiple processors and large memories to speed up the process of creating new pages) from those serving pages. Consequently, response times are not adversely affected around the times of peak updates. By contrast, in early instances of the system, processors functioning as Web servers also performed updates to the underlying data. This early design implementation combined with high cache miss ratios after updates caused response times to be adversely affected around the times peak updates occurred. The more recent design implementation decreases the cost of the installation by reducing processing requirements.

The origin and POD caches can both be configured to manage consistency using expiration times. In this case, our models of request and update characteristics for each page can be used to dynamically assign expiration times. Alternatively, the origin caches can be configured to explicitly accept invalidation and prefetch messages from the origin servers. That way, high degrees of consistency can be maintained in origin caches when data are changing at unpredictable times. The POD caches are less closely coupled to the origin servers and may not be able to accept invalidation or prefetch messages from origin servers.

These sites achieve high availability by using redundant hardware and by serving pages from different sites in different geographic locations containing replicated information. If a server fails, requests are automatically routed to other servers. If an entire site fails, requests can be routed to one of the other sites. The network contains redundant paths to eliminate single points of failure. The sites are designed to handle at least two to three times the expected bandwidth in order to accommodate the high volumes of data should portions of the network fail.

IV. Dynamic Data Request and Update Patterns

In this section, we consider the stochastic properties of the client request patterns and the dynamic page update patterns exhibited at HAWS providing significant dynamic content. A representative sample of some of the results of our analysis based on data from the Nagano site [4] is presented. We note that our system architectures and algorithms have been successfully deployed for caching dynamic Web content at many HAWS since Nagano, and the results presented in this section are also representative of those found at these more recent Web sites. It is important to further note that our results have significant differences with those presented in [5] for the MSNBC Web site. In contrast to such sites where the occurrence of frequent updates is typically rare (with a default replication process of approximately once an hour) [5], the Web sites of interest in our study are more representative of environments in which frequent updates are relatively common as in sporting event and financial Web sites.

A. Entire Dynamic Data Set

We first focus on the requests for and updates to dynamic pages recorded at the Origin servers in the East Coast site and the Tokyo site, noting that the corresponding traffic patterns for the East Coast site are similar in characteristics to those found at the Midwest site. The number of requests and updates for dynamic pages were examined at different time scales. As a representative example, the graphs in Fig. 4 illustrate the aggregate number of requests and updates encountered every 5 min at the East Coast and Tokyo sites. The request and update times have been adjusted to Greenwich Mean Time (GMT). Observe the different scales for the number of requests versus the number of updates. The request patterns plotted are for an entire site whereas the update patterns are for one of the SP2 frames at a site each of which encounters the same set of updates. We note that the request patterns found at different time scales exhibit forms of burstiness that persist over a wide range of time scales, which is consistent in this respect with the results presented in [14] for a different Web site environment. The update patterns

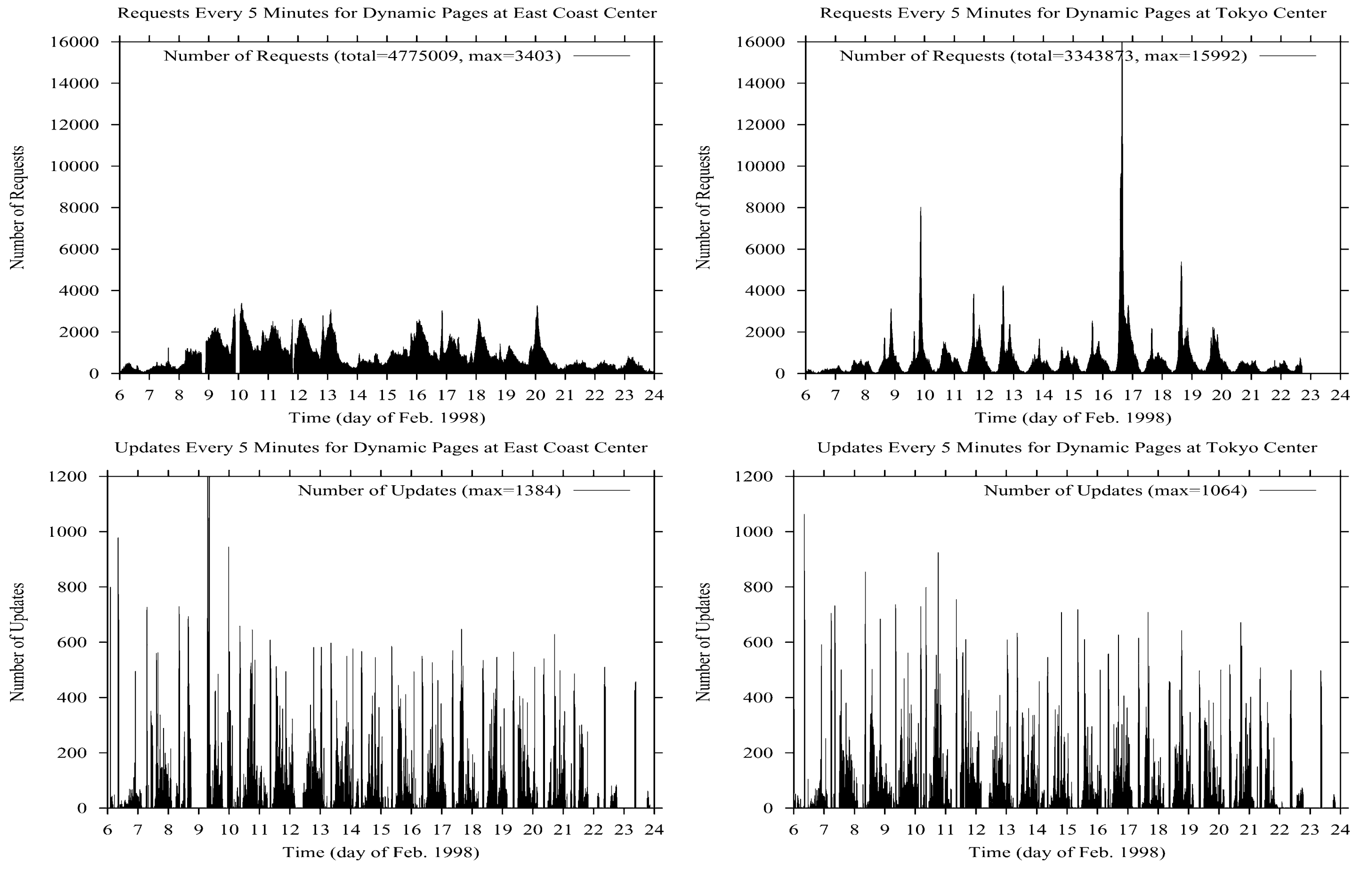

Fig. 4. Request and update patterns every 5 min at the East Coast and Tokyo sites.

similarly exhibit characteristics that persist over a broad range of time scales.

Considering the request traffic, we observe that the patterns at the Tokyo site contain very large bursts of requests and strong periodic patterns, whereas the request traffic found at the East Coast site is much less bursty. Note that the traffic at the Tokyo site primarily served requests originating from Asia, which was essentially dominated by requests from Japan, and that the requests found at the East Coast site primarily originated from Europe. The large bursts in requests from Asia illustrated in Fig. 4 are primarily due to a strong public interest in Japan for events related to ski jumping, and these bursts were concentrated around the time when such popular ski jumping events occurred. In contrast, likely due to the time differences and the more diverse interests in events, the requests from Europe are more scattered and do not contain the very large bursts of requests found in the traffic from Asia. Another possible cause for the smoother request patterns from Europe might be the limited bandwidth for the transatlantic traffic, which could act as a "filter" that smooths out the burstiness. On the other hand, the request patterns at the Midwest site, which served requests primarily originating from the Americas, are quite similar in characteristics to the requests originating from Europe (without any transatlantic bandwidth limits). In any event, these different request patterns would tend to result in somewhat different caching decisions at the East Coast and Tokyo sites; see Section V.

To better understand these aggregate request patterns, we consider some key properties of the corresponding marginal (or empirical) distribution. Let X_n denote the number of requests that arrive to a Web site (or individual server) at the nth time unit, and let $\{X_n\}$ denote the corresponding request process. Upon examining the tail distribution of the request process, we find that the request traffic data from the East Coast site appear to follow a light-tailed distribution, i.e., the tail of the empirical distribution decays at least exponentially fast. Conversely, the request process found at Tokyo appears to have a long-tailed distribution, i.e., the tail of the empirical distribution decays at a rate slower than exponential. More precisely, we have $\log \mathrm{P}[X > x] \sim -\alpha x^2$ for the East Coast site request data and $\log \mathrm{P}[X > x] \sim -\beta(\log x)^2$ for the Tokyo site request data, where $X \overset{d}{=} X_n$ denotes the generic random variable that follows the marginal distribution of $\{X_n\}$.

Since these light-tailed and long-tailed empirical distributions only characterize the marginals of the request processes in terms of the batch size per unit time, we also consider the dependence structure and periodicity of the processes $\{X_n\}$ over time. Here we find that the class of $\mathrm{ARIMA}(p,d,q)$ processes are quite suitable for our purposes, where p defines the order of the autoregressive process, d defines the degree of differencing, and q defines the order of the moving-average process. Periodic, or seasonal, patterns can be additionally captured by extending this class of stochastic processes to the so-called *seasonal ARIMA* processes, denoted by $\mathrm{ARIMA}(p,d,q) \times (P,D,Q)_s$ where P defines the order of the seasonal autoregressive process, D defines the degree of seasonal differencing, Q defines the order of the seasonal moving average process, and

s defines the span of the seasonality. See [15] and [16] for additional details on these and related stochastic processes.

Our analysis of the request process $\{X_n\}$ from the East Coast site demonstrates a strong dependence structure within stationary (or near-stationary) intervals that can be accurately represented by a low-order ARMA process or a low-order seasonal ARMA process. Similarly, the request process from the Tokyo site after taking a log transformation has a strong dependence structure within stationary (or near-stationary) intervals that can be accurately represented by a low-order seasonal ARMA process. There is also consistent nonstationary behavior with daily and weekly patterns that can be accurately captured with the appropriate seasonal versions of these processes. We note that the combinations of the foregoing marginal distributions and dependence structures found in the request patterns at the East Coast and Tokyo sites can significantly degrade system performance over that experienced under more typical arrival processes [17]. We therefore exploit these results in the algorithms and architectures presented in Sections II and III.

It is important to note that in some Web site environments, including HAWS that we have experience with, a form of correlation exists among client requests as a result of an exogenous behavioral process. In the specific case of Nagano [4], various sporting events were taking place at known times which often caused fans of a particular sporting event to request the corresponding pages from the Web site around the time when the results for the sporting event were expected to be available. For example, recall our previous comments about interest in Japan for events related to ski jumping; in fact, the largest burst of requests encountered at the Tokyo site were concentrated around the time when Japan won the gold medal in ski jumping. The exogenous process comprised of these event times, in which a set of users have particular interest, tends to create certain complexities in the dependence structure and periodicity of the client request patterns over time. In some cases, there tend to be additional correlations between the update patterns for a page and the request patterns for the page, possibly due to users reloading the page to obtain the latest results for the corresponding event. Since Nagano consisted of multiple events in different sports taking place concurrently, the traffic patterns encountered at the Web site are impacted by the superposition of multiple instances of such exogenous behavioral processes.

Turning now to the update traffic in Fig. 4, we observe that the patterns at both the East Coast and Tokyo sites contain relatively large bursts of updates and periodic behavior. Note that any completion of an event often caused multiple entries in the database to be modified, each of which often resulted in updates to multiple pages. This was often exacerbated by human error where a subsequent database entry was needed to correct a mistake in the initial entry, all of which helps to explain the relatively large update bursts. The similar update patterns at each set of servers is a result of the algorithms and architectures described in Sections II and III. There are, however, some noticeable differences that are primarily due to servers being taken off line. To simplify early implementations, a straightforward scheme was employed at Nagano where updates were queued when a server went down and then these updates would be serially processed in order when the server came back up. This explains some of the large bursts of updates following a period of no updates in portions of one plot as compared to the other plot (e.g., compare the two update process plots on February 8 and 9). These effects have been eliminated in more recent instances of our system design. Somewhat more minor differences in the two update processes illustrated in Fig. 4 are due to small delays among the servers in receiving and processing updates. This is due in part to the geographically distributed system architectures.

Let U_n denote the number of updates that arrive to a Web site (or individual server) at the nth time unit, and let $\{U_n\}$ denote the corresponding update process. Upon examining the tail distribution of the update process, we find that the update traffic data from both the East Coast and Tokyo sites appear to follow a light-tailed distribution. More precisely, we have $\log \mathrm{P}[U > u] \sim -\gamma u$ for the update data from both sites, where $U \overset{\mathrm{d}}{=} U_n$ denotes the generic random variable that follows the marginal distribution of $\{U_n\}$. Our analysis of the update processes further demonstrates a relatively weak short-range dependence structure and some relatively strong periodic behavior within stationary (or near-stationary) intervals. Here we find that the dependence structure of $\{U_n\}$ can be accurately represented by a seasonal ARMA process.

We next consider the number of requests to individual pages, the number of updates to individual pages, and the CPU overheads for generating these pages over the entire set of dynamic pages at the Nagano Web site [4]. In the interest of space, we provide in Fig. 5 the data collected from one of the three SP2 frames at the East Coast site, noting that these plots are very similar to the corresponding plots from all other frames at the East Coast, Tokyo, and Midwest sites. In Fig. 5, plots (a), (b), and (c) respectively show the total number of requests, the total number of updates, and the CPU overheads for each individual page indexed by the rank of its request frequency. Similarly, plots (d), (e), and (f) respectively show the total number of requests, the total number of updates, and the CPU overheads for each individual page indexed by the rank of its update frequency. The request frequency curve indexed by the request rank appears to be a standard Zipf-like function. In particular, the function $f(x) = 200\,000/x$ provides a reasonably good fit, as illustrated in Fig. 5(a). Fig. 5(b) exhibits somewhat of a pattern where the more frequently requested pages turn out to be updated more frequently (with a number of exceptions). Fig. 5(c) shows that the CPU overheads are relatively independent of the request frequencies. Similar to Fig. 5(b), the request frequency curve indexed by the update rank in Fig. 5(d) also exhibits a decreasing pattern, with a few exceptions. The update frequency curve indexed by the update rank in Fig. 5(e) appears to be reasonably approximated by the function $f(x) = 4000/\sqrt{x}$, which is different from a standard Zipf-like function. It is interesting to note that the CPU overhead curve indexed by the update rank in Fig. 5(f) remains flat for a while and then increases to a much higher level. This suggests that the pages which are frequently updated do not have a large CPU overhead for their updates, which is primarily due to the design of the Web pages and the nature of the site. This is just another aspect of the design that

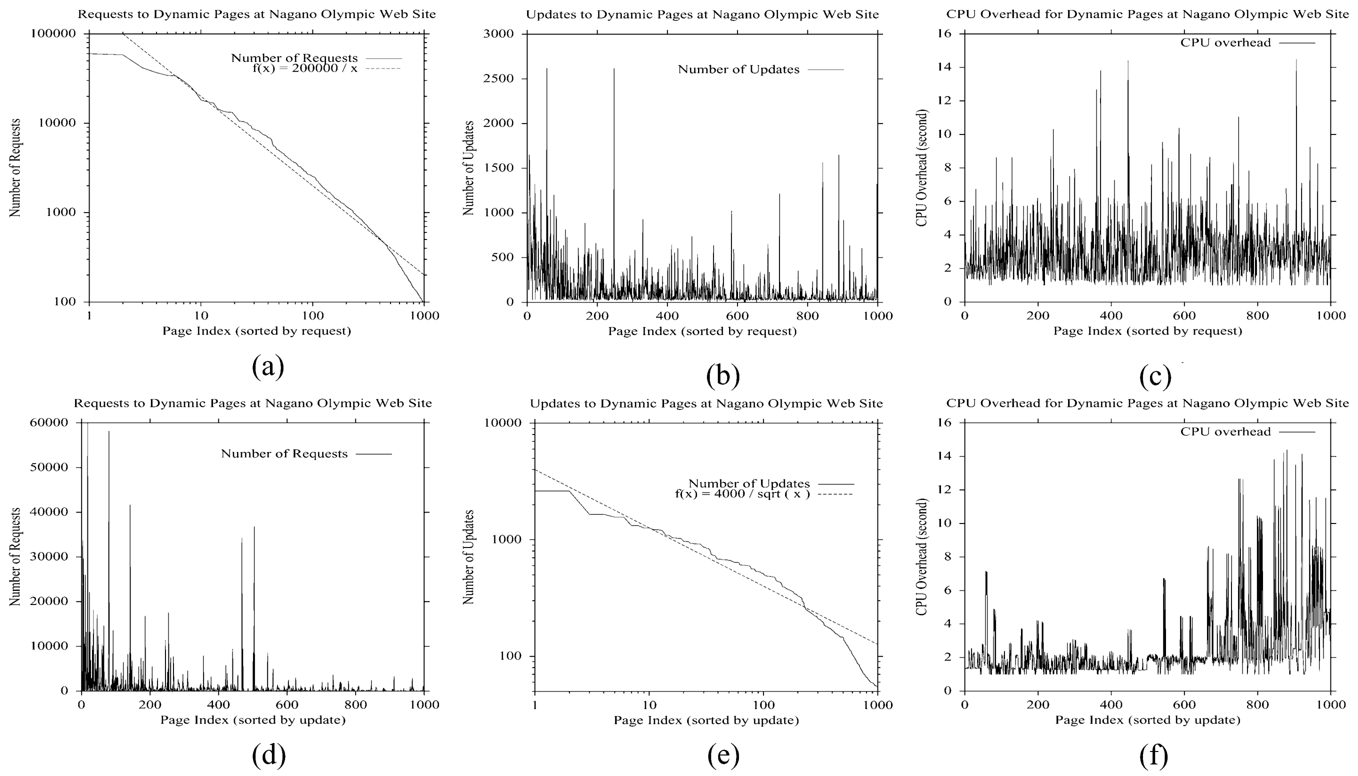

Fig. 5. Request and update distributions to dynamic pages.

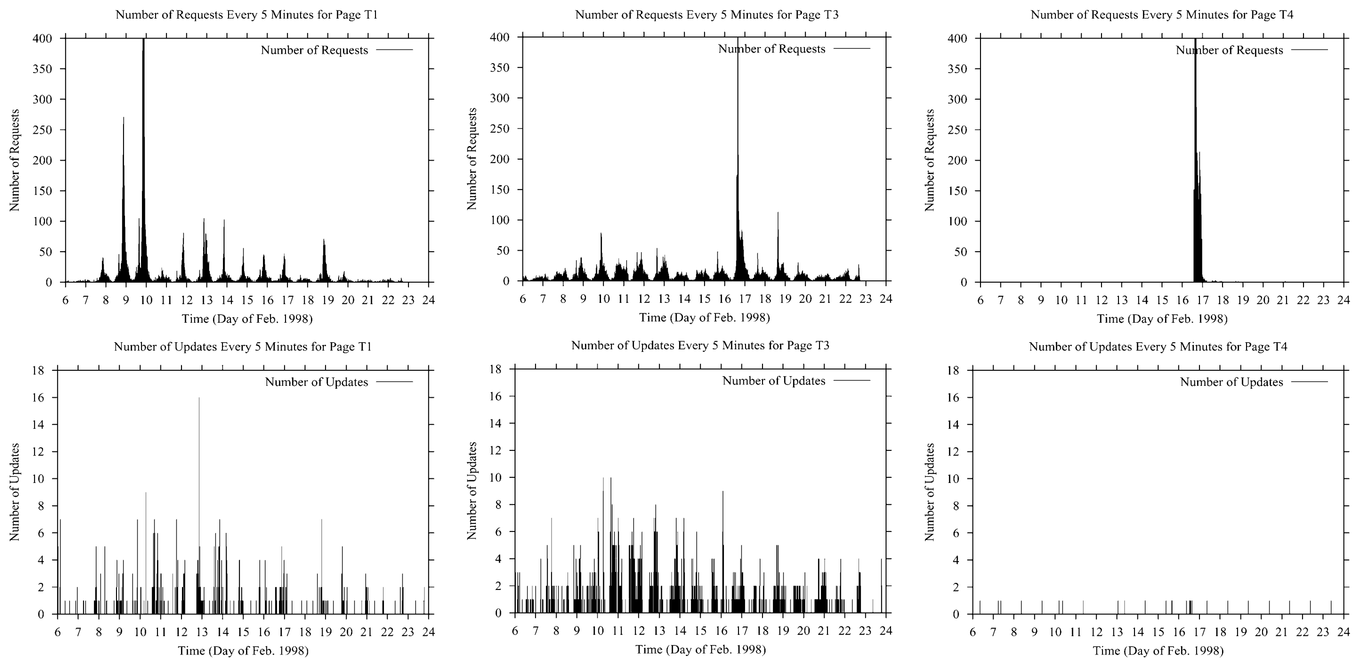

Fig. 6. Requests and updates to the first (T1), third (T3), and fourth (T4) most popular dynamic pages from Asia.

has been and continues to be exploited to maintain good performance at the HAWS.

B. Individual Dynamic Pages

We now consider the request and update patterns for the pages most frequently accessed from Europe and Asia to better understand the request and update patterns for the individual pages and their corresponding impact on the server cache. Fig. 6 plots the request and update processes for three of the four most frequently accessed pages from Asia. (The corresponding results for the second most popular page, as well as the corresponding set of results from Europe, can be found in [18].) Noting the different scales for the number of requests and number of updates, we observe that popular dynamic pages are requested far more frequently than they are updated. In fact, pages for sporting events in progress were updated as often as once or twice per

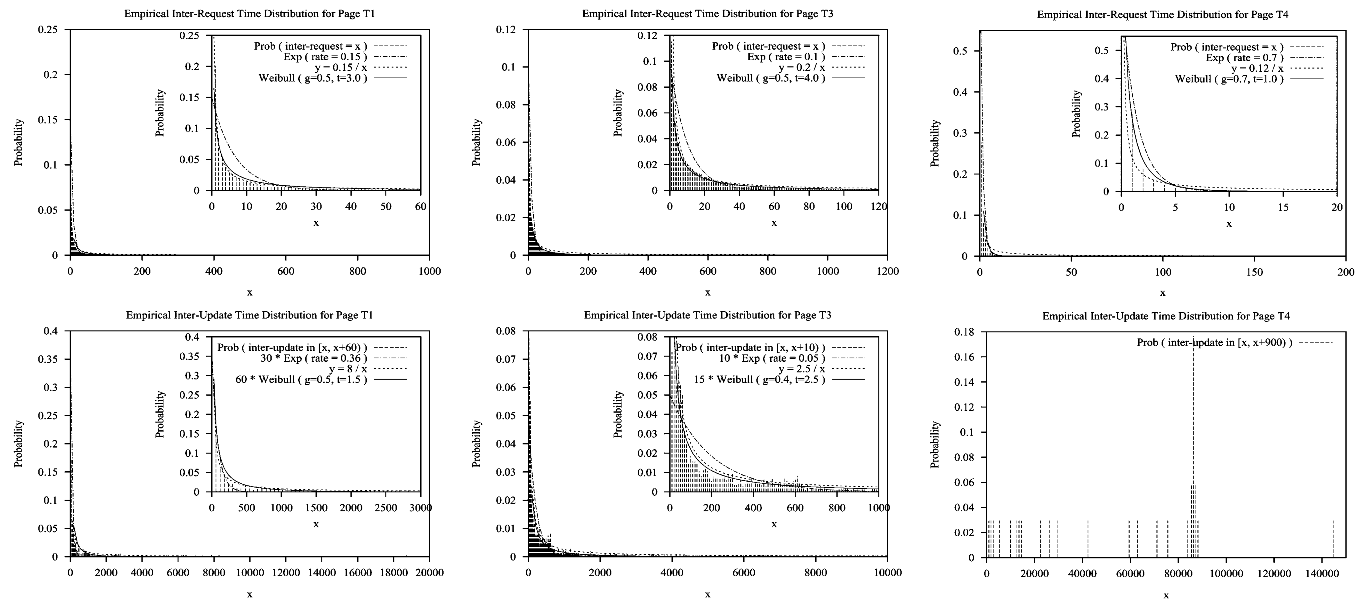

Fig. 7. Interrequest and interupdates times for the first (T1), third (T3), and fourth (T4) most popular dynamic pages from Asia.

minute, whereas requests for these same pages tended to arrive at rates of up to several thousand per second during peak periods. Further observe that both processes are nonstationary as they exhibit fairly strong periodic (daily) patterns. In the case of Europe, we note that the requests to certain pages turn out to be fairly regular while the requests to some other pages turn out to be more bursty. It is interesting to find that some of the pages do not have many updates, and thus caching such pages may provide considerable performance gains given their request patterns. Turning to the case of Asia, we observe that the requests to the first page tend to mimic the request patterns for all of the pages, whereas the requests to the other pages turn out to be very bursty. It is also interesting to observe that (essentially) all of the requests to the fourth most popular page occur at around the same time. Note that this is the page for the team ski jumping event for which the Japanese team won the gold medal, and that the burst of requests are centered around the time of this event. Moreover, the page for the team ski jumping event does not have many updates because its content only changes during the event and at routine daily update times. Clearly, we would like to prefetch such pages into the cache shortly before the event begins, keep it in cache through the course of the event, and then evict the page from the cache somewhat after the event ends. We further observe a certain degree of positive correlations between the requests and updates to the same page. In the case of Asia, the correlation factors for the request and update processes of the four most popular pages are 0.13, 0.18, 0.09, and 0.02, respectively, whereas in the case of Europe, the correlation factors for the request and update processes of the four most frequently accessed pages are 0.16, 0.14, 0.12, and 0.06, respectively. By exploiting such correlations in the caching policy, updates can be used to prefetch certain pages as described in Section II.

To further understand the request and update processes in terms of interrequest and interupdate time statistics, Fig. 7 plots the empirical distributions for the interrequest and interupdate processes of three of the four most frequently accessed pages from Asia. A more detailed view around the origin is embedded within each plot for a clearer illustration of the curve fitting. (The corresponding results for the second most popular page, as well as the corresponding set of results from Europe, can be found in [18].) Note that the interupdate times for a page are precisely the lifetimes for the page. In general, we find that the interrequest time distributions can be closely approximated by the class of Weibull distributions, where the density function has the form $f(x) = (\gamma/\theta)x^{\gamma-1}e^{-x^{\gamma}/\theta}$, for $x > 0$, $\theta > 0$ and $\gamma > 0$. Nevertheless, the Zipf distributions provide a reasonably good fit, as also illustrated in the figure. We observe from these results that the interrequest and interupdate distributions are considerably different from the exponential distribution. Moreover, recall that the request and update processes are nonstationary. Hence, a simple stationary Poisson process is probably not sufficiently accurate for these types of processes encountered at HAWS. It is also interesting to note that the interupdate time, or lifetime, distributions for certain pages are multimodal. This phenomenon is most significant for the second and fourth most popular page from Europe and the third and fourth most popular page from Asia. One possible explanation for this is that the lifetimes are either very short, which means the updates are occurring throughout the day, or very long, which means the updates are separated by long periods such as overnight.

Given the very limited previous results in the research literature on the update patterns found in popular highly dynamic Web sites, we further investigate the tail distribution of the lifetimes for all dynamic pages that are updated relatively often. In particular, plots of the interupdate tail distributions are considered for each of a set of 18 individual dynamic pages that are representative of the update patterns found in our study of all dynamic pages which were modified a considerable amount of time. (These plots are omitted due to space limitations; see [18].) We first observe from these results that the interupdate time distribution for some of the Web pages have a tail that decays at a subexponential rate and can be closely approximated

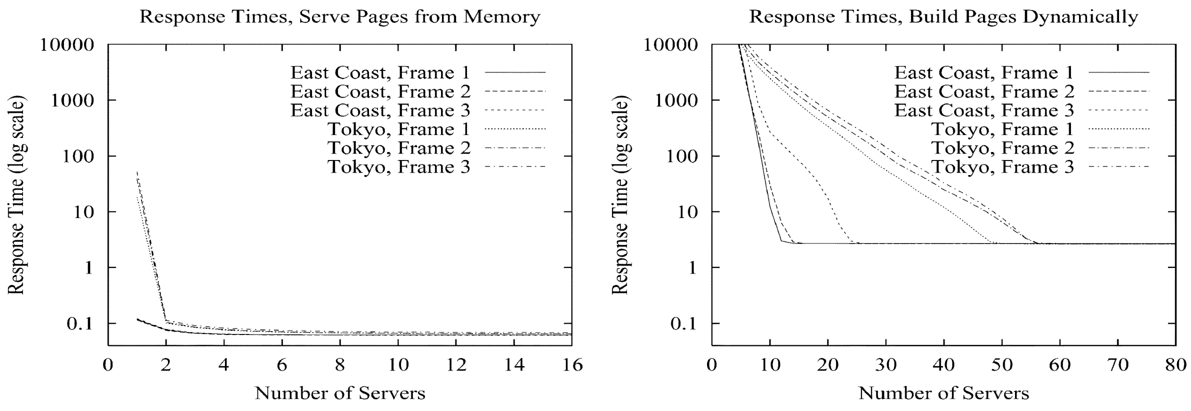

Fig. 8. Mean response times for three representative SP2 frames from the East Coast and Tokyo sites.

by a subset of the Weibull distributions. We further find that the interupdate time distribution for some of the other Web pages have a heavy tail and can be closely approximated by the class of Pareto distributions, where the tail distribution is given by $\overline{F}(t) = (\beta/(\beta + t))^{\alpha}$, for $t \geq 0, \beta > 0$ and $0 < \alpha < 2$.

V. Performance Analysis

We now consider the performance benefits of the algorithms and architectures presented in Sections II and III. A representative sample of some of the results of our analysis using data from Nagano [4] is presented, quantifying the significant performance improvements of our system design.

A trace-driven simulation is used to model an SP2 frame where the input trace is constructed from the access and cache reference logs from the servers comprising the frame. Specifically, for each frame, we merged all of the server access logs over the entire duration of Nagano [4] into a single trace representing the request traffic served by the SP2 frame. We extracted the page updates from the cache reference log for one of the servers in the frame, which was always on-line, and merged this information into the corresponding per-frame trace. This trace is then used as input to the simulator to model the request and update processes for the SP2 frame of interest. We consider each of six such SP2 frames using the corresponding workload trace. This collection of SP2 frames, three from the East Coast site and three from the Tokyo site, are representative of the Nagano Web site as a whole. We further consider a single POD cache for each of the SP2 frames, with different cache sizes as described below.

To complete the workload model for our simulation experiments, we conducted a large set of detailed measurements on an isolated processor in an SP2 frame to obtain the server resource requirements for each page. These measurement-based experiments reveal that the time to serve dynamic page requests from the server memory fits very well to a linear function of the page size, at least for the Web environment of interest. We therefore used in the simulator the corresponding linear function obtained from measurement data to model the server resource requirements for each dynamic page request, served from memory, based on the size of the request provided in the access log. On the other hand, our measurement-based experiments reveal that the time to build dynamic page requests depends upon a number of different factors. We therefore use directly in our simulation model the detailed measurements for the server resource requirements needed to build each dynamic page when the page is not resident in the server memory.

To isolate and compare the different aspects of the architectures of Section III, we first consider the case where POD caches are not exploited and thus all requests go directly to the Origin servers. Note that this was the system architecture employed at Nagano [4]. In Fig. 8, we plot the mean response times (in log scale) as a function of the number of servers under the request patterns found at three representative SP2 frames from the East Coast and Tokyo sites. The leftmost graph represents the cases in which the algorithms and architectures of Sections II and III are exploited, whereas the rightmost graph represents the cases in which the dynamic pages are served using standard techniques. Note that throughput is equal to the mean arrival rate at each of the SP2 frames, since no requests are dropped and the system is work conserving. Thus, throughput is a fixed value independent of the number of servers, and therefore it is not an interesting performance measure to consider in the context of this section. We observe from the results in Fig. 8 that our approach provides significant performance benefits over the standard methods for serving dynamic content, with differences between the best mean response times exceeding an order of magnitude. Our approach also requires significantly fewer servers to achieve the corresponding minimum mean response times than those required under the common approach. By exploiting the DUP algorithms to maintain up-to-date copies of the dynamic pages cached at each server, the system can serve dynamic content at the performance level of serving static content. In particular, for the HAWS, the architectures and algorithms presented in this paper provide more than an order of magnitude improvement in performance using an order of magnitude fewer servers over that observed for the commonly used methods to serving dynamic content.

We next consider the architecture of Section III in which POD cache farms are exploited in front of each site. Recall that this architecture was subsequently employed at HAWS following Nagano [4]. In Fig. 9, we plot the mean response times (in log scale) and the corresponding cache request hit ratios as a function of the number of servers under the request patterns found at three representative SP2 frames from the East Coast and Tokyo sites. Once again, the leftmost graphs represent the

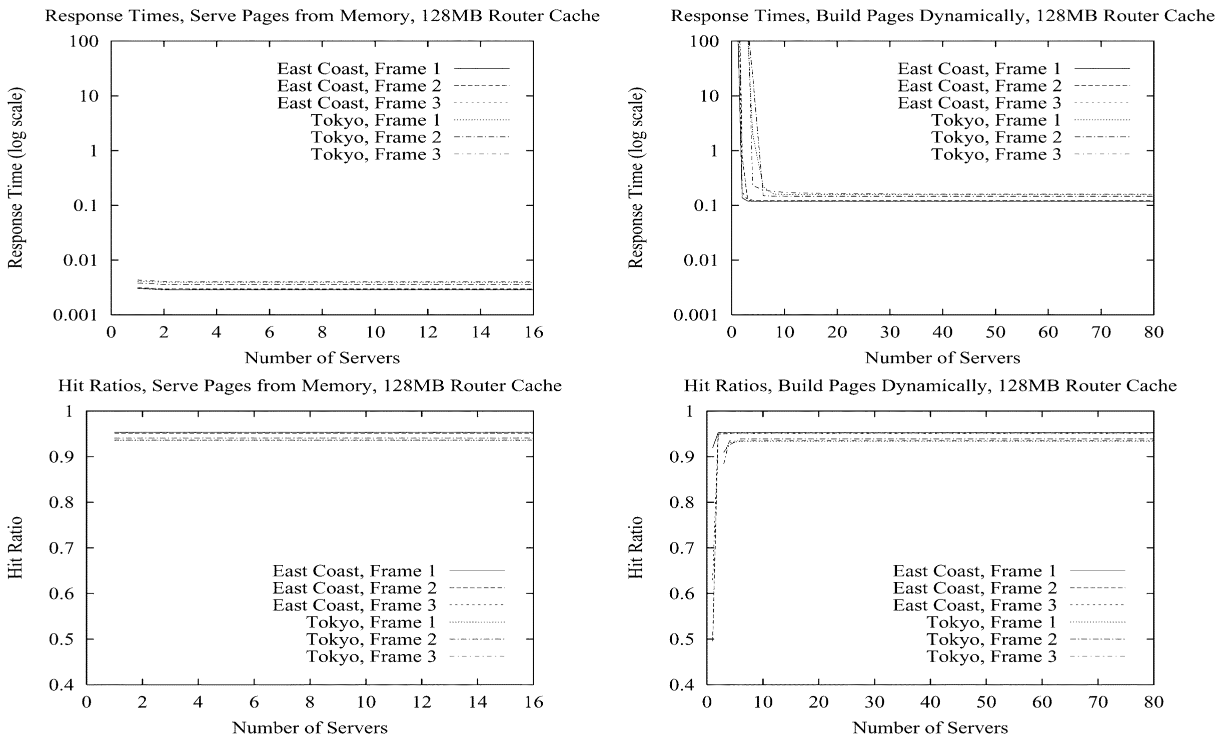

Fig. 9. Mean response times and cache hit ratios for three representative SP2 frames from the East Coast and Tokyo sites, each with a POD cache farm.

cases in which our architectures and algorithms are exploited, whereas the rightmost graphs represent the cases in which standard methods are used together with POD cache farms. The response time measures are provided in the uppermost graphs and the hit ratio measures are provided in the lowermost graphs. In each case, the presented results are for a POD cache of size 128 MB. Note that the corresponding results for larger cache sizes are identical to those provided in Fig. 9, as the content data set at Nagano was around 100 MB.

We observe from these results that the POD cache farms provide significant performance gains under both approaches to serving dynamic content, where the best mean response times are reduced by more than an order of magnitude in each case. There also is a significant reduction in the number of servers needed to achieve the corresponding minimum mean response times under the standard methods, with a much more modest reduction under our approach (i.e, from a very few servers to a single server). We continue to observe that our approach provides more than an order of magnitude improvement in performance over that obtained under the commonly used approach. Moreover, the mean response times curves under our approach are essentially flat, whereas the mean response times under standard methods tend to increase dramatically when there are less than a few servers. Similarly, the cache hit ratio curves are flat under our approach, ranging from above 93% to below 96%. In general, the corresponding maximum hit ratios under the commonly used methods are slightly smaller when there are a sufficient number of servers, and much smaller when too few servers are employed. Finally, with the exception of insufficient servers, the common approach together with the POD cache farms yield performance results that are much closer to those under our approach without the cache farms, although the latter continues to provide considerably lower mean response times (recall the log scale).

Given the relatively small content data set at Nagano [4], we also investigate the architectures of Section III with POD cache farms under workloads having inflated data sets that cover the full range of content data set sizes found at HAWS we have experience with. In particular, we performed simulation experiments under the request patterns of the three representative SP2 frames from the East Coast and Tokyo sites in which the size of each page is increased by a multiplicative factor. The corresponding mean response times and cache hit ratios for an inflation factor of 20 and POD cache sizes of 128 MB, 256 MB, 512 MB, and 1 GB are considered as a function of the number of servers. (These plots are omitted due to space limitations; see [18].) In comparison with Fig. 9, we observe that the larger content data set causes significant degradation in performance under both approaches, as expected. The largest performance improvements are seen under both approaches as the cache size is increased from 128 to 256 MB, with a somewhat smaller improvement as the cache size is increased from 256 to 512 MB. An even smaller relative improvement in performance is observed as the cache size is increased from 512 MB to 1 GB.

While the POD cache farms provide significant performance gains under both approaches, we note that the architectures and algorithms presented in this paper yield considerable improvements in performance over that obtained under the commonly used approach for serving dynamic content even for workloads with much larger content data sets than those used at Nagano [4]. There also is a considerable reduction in the number of servers needed to achieve this level of performance under our approach. Moreover, by exploiting the DUP algorithms to maintain up-to-date copies of the dynamic pages cached at each server, the POD caching policies can be simplified, and performance can be further improved as all levels of the system architectures serve dynamic content at the performance level of serving static content.

Finally, it is important to further note that our experience at Nagano [4] and other subsequent HAWS is consistent with the results of the foregoing performance analysis. In particular, system measurements based on Nagano confirm that the architectures and algorithms presented in this paper provide more than an order of magnitude improvement in performance using an order of magnitude fewer servers over that obtained via standard methods. Moreover, system measurements from the Web site architectures of Section III demonstrate an additional order of magnitude improvement in performance with each POD cache able to handle the equivalent of roughly 10 server nodes. Furthermore, the cache hit ratios observed in practice for different data set sizes are consistent with the corresponding results presented above. See [13] for additional details on our analysis of the caching effects for the Sydney site [6].

VI. Related Work

Zhu and Yang developed a system for caching dynamic Web content in which data dependencies are managed at a coarse level of granularity [19]. Dynamic pages are partitioned into classes based on URL patterns so that an application can specify page identification and data dependence, and invoke invalidation for a class of dynamic pages. This is analogous to a caching system we used for the 1996 Olympic Games [10] in which caches were organized by sport and invalidation was performed on a sport-by-sport basis [1]. We found that the use of fine grained dependencies for precise updates and invalidations improved cache hit ratios considerably over approaches based on coarse grained dependencies. Holmedahl, Smith, and Yang developed a system for cooperatively caching the results of requests for dynamic content [20]. In their system, cached objects are invalidated based on expiration times, so good performance can only be achieved if expiration times can be accurately predicted in advance. Smith, Acharya, Yang, and Zhu [21] developed a protocol to allow individual content-generating applications to exploit query semantics and specify how their results should be cached and/or delivered.

Cao, Zhang, and Beach proposed a scheme for the caching of dynamic documents at Web proxies [22]. Their method migrates parts of server processing to caching proxies using cache applets which are server-supplied code that is attached to URLs or collections of URLs. Douglis, Haro, and Rabinovich proposed an extension to HTML in order to support dynamic document caching [23]. Their approach separates static and dynamic portions of a resource. The static portion can be cached. Vahdat and Anderson [24] proposed a scheme for caching dynamic Web content which updates cached data by profiling the execution of programs creating dynamic Web content and determining when an input file changes. This approach is only applicable to a limited class of dynamic Web content and is not sufficient for dynamic content generated from database queries.

It is becoming increasingly common to generate Web pages from fragments [7], [25], [26]. Dynamic and personalized parts of the Web pages can be encapsulated within fragments which are not cached, while other more static fragments may be cached. This method significantly improves performance of dynamic content. DUP can be used in conjunction with fragment-based Web page generation to maintain consistency.

Crovella and Bestavros [14] and Arlitt and Williamson [27] both investigate different aspects of Web traffic patterns at the server level, including interrequest times with heavier than exponential tails and heavy-tailed file-size distributions. Arlitt and Jin analyze the traffic from the 1998 World Cup Web site [28]. Liu, Niclausse, and Jalpa-Villanueva investigate the characteristics of HTTP requests at the session level [29]. Arlitt, Krishnamurthy, and Rolia [3] analyze five days of workload data from a large Web-based shopping system to assess scalability and support capacity planning. Our study complements this previous work by providing an analysis of a different Web server environment in which the density of requests is significantly higher, with the exception of [28]. Moreover, in comparison with the world-wide soccer event in [28], the Olympic Games considered in this paper consists of multiple events in different sports that take place simultaneously.

Our study is one of the first to investigate the corresponding update process encountered at HAWS serving dynamic content, together with Padmanabhan and Qiu [5] who investigate the request and update patterns of dynamic pages at the MSNBC Web site. There are, however, considerable differences between our results and those in [5], due in part to the significantly more frequent updating of the dynamic content in the Web sites of interest in our study, and we further exploit the results of our analysis of the update and request patterns in the design of architectures and algorithms to reduce the overhead of serving dynamic content.

VII. Conclusion

In this paper, we have presented general multitier architectures and a set of algorithms that have been deployed for high-performance serving of dynamic content to many clients at HAWS. This system design is based on our analysis of the properties of the request and update patterns at HAWS environments and on our analysis of the performance properties of the architectures and algorithms.

Analysis of our system design based on data and measurements from real deployments at HAWS demonstrate significant performance benefits, which has proven to be consistent with our considerable experience using this system design at subsequent HAWS. In particular, our system is able to achieve cache hit ratios close to 100% compared with 80% for an earlier version of our system which did not use these architectures and algorithms. Proper deployments result in cached data which is almost never obsolete by more than a few seconds, if at all. Moreover, this system design serves dynamic content at the performance level of static content.

Much of the data in this paper was obtained by deployment of our system at sports and special event Web sites. We have also successfully deployed our system at financial Web sites providing current stock quotes. At financial Web sites, the rate at which updates occur is even higher. However, the number of Web pages affected by a particular update tends to be lower.

REFERENCES

[1] A. Iyengar and J. Challenger, "Improving Web server performance by caching dynamic data," in *Proc. USENIX Symp. Internet Technologies and Systems*, Dec. 1997, pp. 49–60.

[2] A. Iyengar, E. MacNair, and T. Nguyen, "An analysis of Web server performance," in *Proc. IEEE GLOBECOM*, Nov. 1997, pp. 1943–1947.

[3] M. Arlitt, D. Krishnamurthy, and J. Rolia, "Characterizing the scalability of a large Web-based shopping system," *ACM Trans. Internet Technology*, vol. 1, no. 1, pp. 44–69, Aug. 2001.

[4] 1998 Nagano Olympic Games Web Site. [Online]. Available: http://www.nagano.olympic.org

[5] V. N. Padmanabhan and L. Qiu, "The content and access dynamics of a busy Web site: Findings and implications," in *Proc. ACM SIGCOMM*, 2000, pp. 111–123.

[6] 2000 Sydney Olympic Games Web Site. [Online]. Available: http://www.olympics.com

[7] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed, "A publishing system for efficiently creating dynamic Web content," in *Proc. IEEE INFOCOM*, Mar. 2000, pp. 844–853.

[8] J. Challenger, A. Iyengar, and P. Dantzig, "A scalable system for consistently caching dynamic Web data," in *Proc. IEEE INFOCOM*, Mar. 1999, pp. 294–303.

[9] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[10] 1996 Atlanta Olympic Games Web Site. [Online]. Available: http://www.atlanta.olympic.org

[11] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig, "High-performance Web site design techniques," *IEEE Internet Computing*, vol. 4, no. 2, pp. 17–26, Mar./Apr. 2000.

[12] J. Song, A. Iyengar, E. Levy, and D. Dias, "Architecture of a Web server accelerator," *Comput. Networks*, vol. 38, no. 1, pp. 75–97, 2002.

[13] Z. Liu, M. S. Squillante, C. Xia, S. Yu, L. Zhang, N. Malouch, and P. Dantzig, "Analysis of caching mechanisms from sporting event Web sites," in *Advances in Computing Science—ASIAN 2002*, A. Jean-Marie, Ed. New York: Springer-Verlag, 2002.

[14] M. E. Crovella and A. Bestavros, "Self-similarity in World Wide Web traffic: Evidence and possible causes," *IEEE/ACM Trans. Networking*, vol. 5, pp. 835–845, Dec. 1997.

[15] P. J. Brockwell and R. A. Davis, *Time Series: Theory and Methods*. New York: Springer-Verlag, 1987.

[16] M. Kendall and J. Ord, *Time Series*. Oxford, U.K.: Oxford Univ. Press, 1990.

[17] M. S. Squillante, D. D. Yao, and L. Zhang, "Web traffic modeling and server performance analysis," in *Proc. 38th IEEE Conf. Decision and Control (CDC)*, 1999, pp. 4432–4439.

[18] J. Challenger, P. Dantzig, A. Iyengar, M. S. Squillante, and L. Zhang, "Efficiently serving dynamic data at highly accessed Web sites," IBM Research Div., Yorktown Heights, NY, Tech. Rep. RC22823, 2003.

[19] H. Zhu and T. Yang, "Class-based cache management for dynamic Web content," in *Proc. IEEE INFOCOM*, Apr. 2001, pp. 1215–1224.

[20] V. Holmedahl, B. Smith, and T. Yang, "Cooperative caching of dynamic content on a distributed Web server," in *Proc. 7th IEEE Int. Symp. High Performance Distributed Computing*, July 1998, pp. 243–250.

[21] B. Smith, A. Acharya, T. Yang, and H. Zhu, "Exploiting result equivalence in caching dynamic Web content," in *Proc. 2nd USENIX Symp. Internet Technologies and Systems*, Oct. 1999.

[22] P. Cao, J. Zhang, and K. Beach, "Active cache: Caching dynamic contents on the Web," in *Proc. Middleware 1998*, pp. 373–388.

[23] F. Douglis, A. Haro, and M. Rabinovich, "HPP: HTML macro-preprocessing to support dynamic document caching," in *Proc. USENIX Symp. Internet Technologies and Systems*, Dec. 1997, pp. 83–94.

[24] A. Vahdat and T. Anderson, "Transparent result caching," in *Proc. 1998 Annu. USENIX Tech. Conf.*, June 1998, pp. 25–37.

[25] P. Mohapatra and H. Chen, "A framework for managing QoS and improving performance of dynamic Web content," in *Proc. IEEE GLOBECOM*, Nov. 2001, pp. 2460–2464.

[26] 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," in *Proc. ACM SIGMOD*, June 2002, pp. 97–108.

[27] M. F. Arlitt and C. L. Williamson, "Internet Web servers: Workload characterization and performance implications," *IEEE/ACM Trans. Networking*, vol. 5, pp. 631–645, Oct. 1997.

[28] M. Arlitt and T. Jin, "Workload characterization of the 1998 World Cup Web Site," HP Laboratories, Palo Alto, CA, Tech. Rep. HPL-1999-35(R.1), 1999.

[29] Z. Liu, N. Niclausse, and C. Jalpa-Villanueva, "Traffic model and performance evaluation of Web servers," *Perform. Eval.*, vol. 46, pp. 77–100, Oct. 2001.

James R. Challenger received the B.S. degree in mathematics and the M.S. degree in computer science from the University of New Mexico, Albuquerque, in 1976 and 1978, respectively.

He is a Senior Technical Staff Member with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, where he does research on problems related to very highly scaled distributed computing, including high-volume Web serving and scaling problems related to grid computing.

Paul Dantzig received the M.S. degree in computer science and computer engineering from Stanford University, Stanford, CA.

He is a Senior Technical Staff Member and Manager of High Volume Web Serving at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY. He is also a part-time Professor of computer science at Pace University, New York. Recently, he has been particularly interested in teaching about the best practices of architecting and designing high-volume Web sites.

Arun Iyengar (SM'00) received the Ph.D. degree in computer science from the Massachusetts Institute of Technology, Cambridge.

He does research and development into Web performance, caching, storage allocation, and distributed computing at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

Dr. Iyengar is the Chair of the IEEE Computer Society's Technical Committee on the Internet, the Chair of IFIP Working Group 6.4 on Internet Applications Engineering, the National Delegate representing the IEEE Computer Society to IFIP Technical Committee 6 on Communication Systems, and an IBM Master Inventor. He is a member of the Association for Computing Machinery (ACM).

Mark S. Squillante (F'01) received the Ph.D. degree from the University of Washington, Seattle.

He is a Research Staff Member at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, where he works in the Mathematical Sciences Department. His research interests concern the mathematical analysis, modeling, and optimization of the design and control of stochastic systems.

Dr. Squillante is currently serving as an Associate Editor of *Operations Research* and an Editor of *Performance Evaluation*. He is a member of the Association for Computing Machinery (ACM).

Li Zhang (M'99) received the Ph.D. degree from Columbia University, New York, NY.

He is a Research Staff Member at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, where he works in the Distributed Computing Department. His research interests concern the modeling, analysis, and optimization of problems in computer systems and network applications.

Dr. Zhang is a member of the Association for Computing Machinery (ACM).