RC22823 (W0306-090) June 18, 2003
Computer Science

IBM Research Report

Efficiently Serving Dynamic Data at Highly Accessed Web Sites

James R. Challenger, Paul M. Dantzig, Arun K. Iyengar, Mark S. Squillante, Li Zhang
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center ,
P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Efficiently Serving Dynamic Data at Highly Accessed Web Sites

Jim Challenger, Paul Dantzig, Arun Iyengar, Mark S. Squillante, Li Zhang

IBM Research Division

Thomas J. Watson Research Center

Yorktown Heights, NY 10598

Abstract

This paper presents techniques for efficiently serving dynamic data at highly accessed Web sites and presents a quantitative analysis of the design motivation and performance benefits of these techniques. To reduce the overhead for generating dynamic data, we use the Data Update Propagation (DUP) 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. DUP maintains data dependence information between cached objects and the underlying data affecting their values in a graph. When the system becomes aware of a change to the underlying data, graph traversal algorithms are applied to determine which cached objects are affected by the change. Cached objects that are found to be highly obsolete, relative to requests for the page, are then either invalidated or updated depending upon request and update patterns for the objects. DUP is one of a few critical components of a general multitier architecture that has been deployed for high-performance serving of dynamic data to many clients at several Web sites hosted by IBM. The design of this architecture and the DUP algorithms is based on a detailed analysis of the stochastic aspects of the client request patterns and the dynamic page update patterns at a few previous highly accessed Web sites. Our considerable experience with this system design at subsequent highly accessed Web sites has proven to be consistent with the results of this motivating analysis. In particular, our system design is able to achieve cache hit ratios close to 100%, as a result of which Web sites based on our system design are able to serve data quickly even during peak request periods. Proper deployments of our system result in cached data which is almost never obsolete by more than a few seconds, if at all. Our system architecture and algorithms for efficiently serving dynamic data provide more than an order of magnitude improvement in performance using an order of magnitude fewer servers over that obtained under existing methods for serving dynamic content. High availability is achieved via redundant hardware and intelligent routing techniques.

1 Introduction

Many Web sites provide information that is constructed by programs which execute at the time requests are made. Such information is known as dynamic content, in contrast to static content served from files. Dynamic content is important for Web sites that provide rapidly changing information. For example, sports Web sites need to provide the latest information about sporting events, and financial Web sites need to provide current information about stock prices. If pages for such Web sites are generated dynamically by a server program, the server program can return the most recent version of the dynamic content. On the other hand, if files are created to serve the pages statically, it may not be feasible to keep the files current. 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 need for, and the benefits of, dynamic pages is 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 [15, 16, 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.

In addition, it is important for dynamic content caches to provide interfaces that allow an application program to explicitly add, delete, and update cached objects [8]. Explicit management of the cache is essential for optimal performance and consistency. However, interfaces for explicitly managing the contents of caches are not sufficient for achieving both good performance and cache consistency. One of the problems we have encountered at real Web sites

was obtaining good performance after the system received new information. It was difficult to precisely identify which cached pages had changed as a result of the new information. In order to insure that all stale pages are invalidated, many current pages may also be invalidated. This can cause high miss ratios after the system receives new information [15].

In order 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. A frequently requested page might preferably be prefetched into caches immediately after it changes, depending upon its update frequency. A less frequently requested page might be invalidated after it changes and added to a cache on demand after a subsequent request for the page. Our DUP algorithms significantly reduce the number of cached pages that need to be invalidated or updated after new information is received. DUP has been a critical component of a general multitier architecture deployed for high-performance serving of dynamic content to many clients at several highly accessed Web sites hosted by IBM. Whenever new content became available to the computers implementing such a Web site, updated Web pages reflecting these changes are made available to the rest of the world within a minimal amount of time (typically on the order of 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, and other information. The degree of consistency we achieve is sufficient for a large variety of Web sites providing information including those providing news stories, stock quotes, and sports results. Our system architecture and algorithms are not sufficient for all situations, however. It is designed to be lighter weight than transactional systems guaranteeing strong consistency, and thus our system design is not appropriate for transactional systems requiring strong consistency. High availability is achieved via redundant hardware and intelligent routing techniques.

The design of our system architecture and algorithms is based on a detailed analysis of the stochastic aspects of the client request patterns and the dynamic page update patterns at a few previous highly accessed Web sites providing significant dynamic content, including the 1998 Nagano Olympic Games Web site [27] (henceforth referred to as *Nagano*). This is important because the optimal solution to the problems being addressed depends heavily upon the request and update patterns for each page, as noted above. In addition to helping us make system design decisions, our analysis is exploited in the on-line algorithms to dynamically adjust real-time policy decisions for caching, prefetching and invalidation. This system architecture and algorithms have been successfully deployed for caching dynamic content at many subsequent highly accessed sporting and event Web sites, including those from the 2000 Sydney Olympic Games [28] (henceforth referred to as *Sydney*) up to the most recent Web sites for the US Open Tennis, Wimbledon, Australian Open, French Open, Master's Golf and Ryder Cup. The system design has also been successfully deployed at various financial Web sites. Our considerable experience with this system design at these highly accessed Web sites has proven to be consistent with the results of our quantitative analysis.

Given the strong connection between our previous analysis and our design motivation and decisions, this paper also presents a representative sample of some of the results of our stochastic analysis. In particular, we identify the characteristics of stochastic processes that can be used to model the request and update patterns for the dynamic

content served at the Nagano Web site [27]. As noted in [29], the content dynamics is important to study and report in the research literature because of its deep implications on the effectiveness of Web caching mechanisms. In addition to exploiting these models in our system architecture and algorithms, the results of our analysis are of interest in their own right especially given the very limited previous research in the literature on update 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. Our analysis also demonstrates that many highly accessed pages are requested far more frequently than they are updated. 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 and model this information for each page and exploit this collection of information throughout our system for efficiently serving dynamic content. It is important to note that our results have significant differences with those presented in [29] 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) [29], 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.

We also present another aspect of our quantitative analysis upon which the design of our system architecture and algorithms is based. In particular, we investigate the performance benefits of our system design for efficiently serving dynamic content at a representative previous IBM Sport and event Web site, thus quantifying the significant improvements provided by our approach. A detailed simulation model is used together with traces from the Nagano Web site [27] and together with measurements performed as part of our study to obtain various model parameters. The results of our analysis show that the algorithms and architecture 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 existing 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 analysis is consistent with the corresponding results from subsequent deployment of the system design at more recent highly accessed sporting and event Web sites.

The remainder of the paper is structured as follows. Section 2 presents our DUP algorithms for reducing the overhead of delivering dynamic content. Section 3 presents our system architecture for serving dynamic content to large numbers of clients. Section 4 presents an analysis of the stochastic properties of the request and update patterns

upon which our system and algorithm design is based, together with a performance analysis of this highly accessed Web site presented in Section 5. Section 6 discusses related work. Finally, Section 7 briefly summarizes the main results and conclusions of our study.

2 Reducing Dynamic Data Overhead

2.1 Caching

Caching is a critical technique for improving Web performance. A cache stores information so that repeated requests for the same object can obtain the object from the more efficient cache instead of fetching it from a remote source or executing a program to create it. Caching has been successfully deployed for static Web content. Dynamic objects are harder to cache because they change frequently.

A key problem with caching dynamic pages is determining what pages should be cached and when a cached page has become obsolete. We developed the data update propagation algorithm to solve this and related problems. Despite the higher update rates for dynamic content, our detailed analysis of the request and update patterns found at Sport and Event Web sites shows that the number of requests for popular dynamic pages far exceeded the number of updates to those pages; refer to Section 4. Hence, judicious use of caching can significantly reduce the number of times such pages have to be regenerated by a server.

2.1.1 The Data Update Propagation Algorithms

Data update propagation (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, 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 (see Section 4), 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 Figure 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 Figure 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 5 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.

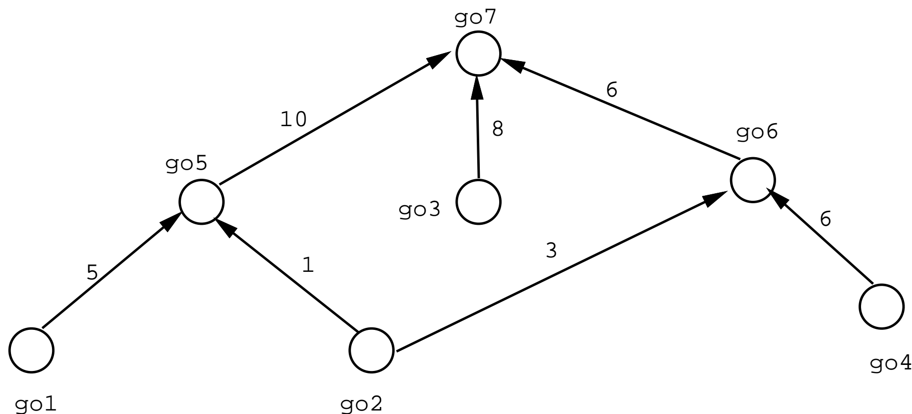

Figure 1: An object dependence graph (ODG). Weights are correlated with the importance of data dependencies.

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. Additional details on the use of these models are provided below, whereas more details on the types of models used for this purpose can be found in Section 4.

In some of the cases we have encountered, the object dependence graph is a *simple object dependence graph* having the following characteristics:

- Each vertex representing underlying data does not have an incoming edge;
- Each vertex representing an object does not have an outgoing edge;
- All vertices in the graph correspond to underlying data (nodes with no incoming edges) or objects (nodes with no outgoing edges);
- None of the edges have weights associated with them.

Figure 2 depicts a simple ODG. We first explain how DUP works for simple ODGs. We then discuss how DUP can be generalized to any ODG.

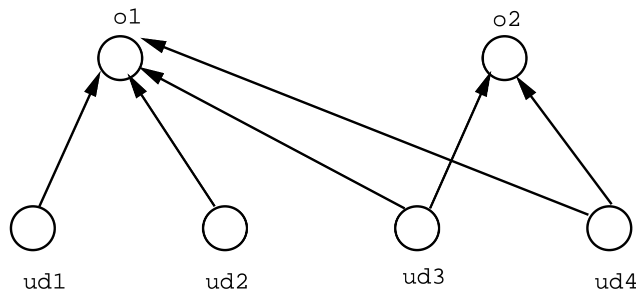

Figure 2: A simple object dependence graph.

2.1.1.1 DUP for Simple Object Dependence Graphs

The application program must determine an appropriate correspondence between underlying data and vertices of the object dependence graph G. For example, a vertex corresponding to underlying data may represent a database table. Another vertex corresponding to underlying data may represent portions of several database tables. There are

no restrictions on how underlying data may be correlated with nodes of G. The application program has freedom to pick the most logical and/or efficient system.

Each object has a string *obj_id* known as the object ID that identifies the object. Similarly, each node representing underlying data has a string *ud_id* known as the underlying data ID which identifies it. The application program informs cache managers that an object has a dependency on underlying data via an application program interface (API) function:

$$add_dependency(obj_id, ud_id).$$

Whenever underlying data corresponding to a node in G changes, an application program notifies cache managers via an API function:

$$underlying_data_has_changed(ud_id).$$

The cache managers then invalidate all cached objects having dependencies on ud_id. Referring to Figure 2,

$$underlying_data_has_changed(ud4)$$

would cause $o1$ and $o2$ to be invalidated. The function call

$$underlying_data_has_changed(ud2)$$

would cause $o1$ to be invalidated.

2.1.1.2 Generalizing DUP to Arbitrary Object Dependence Graphs

2.1.1.2.1 Overview

We now present the generalized DUP algorithms which provide a number of enhancements over the version just presented:

- The generalized DUP algorithms are applicable when the ODG is not simple. In general, a node may have both incoming and outgoing edges as in Figure 1. It is also possible for a graph to have cycles. A cycle would imply that a change to any node in the cycle would result in a change to all nodes comprising the cycle.
- 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 Figure 1. Moreover, the information required to dynamically maintain the request and update models must be tracked.

- 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.

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:

1. Both $v1$ and $v2$ are current.

2. At some point in the past, both versions were current.

2.1.1.2.2 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 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 $n2$.

2.1.1.2.3 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 [1].

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.

2.1.2 Prefetching Pages

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 requested far more frequently than they are updated, as demonstrated by our detailed analysis of the request and update patterns found at Sport and Event Web sites; refer to Section 4. In particular, our analysis shows 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. Similarly, as our analysis in Section 4 further demonstrates, 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 [26], we simply invalidated cached pages when the data changed, relying on demand to cause a cache miss and rebuild the page. Since most pages take 500 to 2000 milliseconds 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 IBM Sport and Event Web sites, whenever data changed, we used the DUP algorithms to first identify the pages affected by the data change. The appropriate pages were re-generated 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 in the system, 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.

2.2 Interfaces for Creating Dynamic Web Data

The interface for invoking server programs that create dynamic pages also has a significant effect on performance. The Common Gateway Interface (CGI) is perhaps 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. Web servers such as Apache, the IBM Go server, and those by Netscape and Microsoft provide APIs for invoking server programs which incur significantly less overhead than CGI. Instead of forking off new processes each time a server program is invoked, server programs are dynamically loaded by the Web server (e.g., IBM's Go Web server API (GWAPI), the Netscape server application programming interface (NSAPI), and Microsoft's Internet Server API (ISAPI) interfaces) or statically linked (Apache) as part of the Web server's executable image. Server programs then execute as part of the Web server's process. There are drawbacks to this approach. Server programs could crash a Web server or leak resources such as memory. In addition, server programs must often be thread-safe; this can complicate the development process considerably.

The second approach, used by Open Market's FastCGI, is to run server programs as long-running processes with which a Web server communicates [23]. This avoids some of the problems of running server programs as part of the Web server's process at the cost of a slightly slower interface which is still considerably faster than CGI.

3 Architecture for Serving Dynamic Data

We now describe the architecture for many of the IBM Sport and Event Web Sites (including the Web sites for the US Open Tennis, Wimbledon, Australian Open, French Open, Master's Golf, Ryder Cup, Nagano and Sydney, Tonys, Grammys, among others) that have served dynamic data to very large numbers of clients. The Web sites typically utilize three to four IBM SP2 multiprocessor systems each containing a maximum of about 125 processors, over 100 GBytes of memory, and several terabytes of disk space. Three of these sites are permanently located in the USA; see Figure 3. A fourth site was added in Japan (specifically, Tokyo) for Nagano [27] and in Australia for Sydney [28] 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 one 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 [31] 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 while the events are taking place 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 a few seconds; as noted above, this time interval tolerance can be adjusted to satisfy 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.

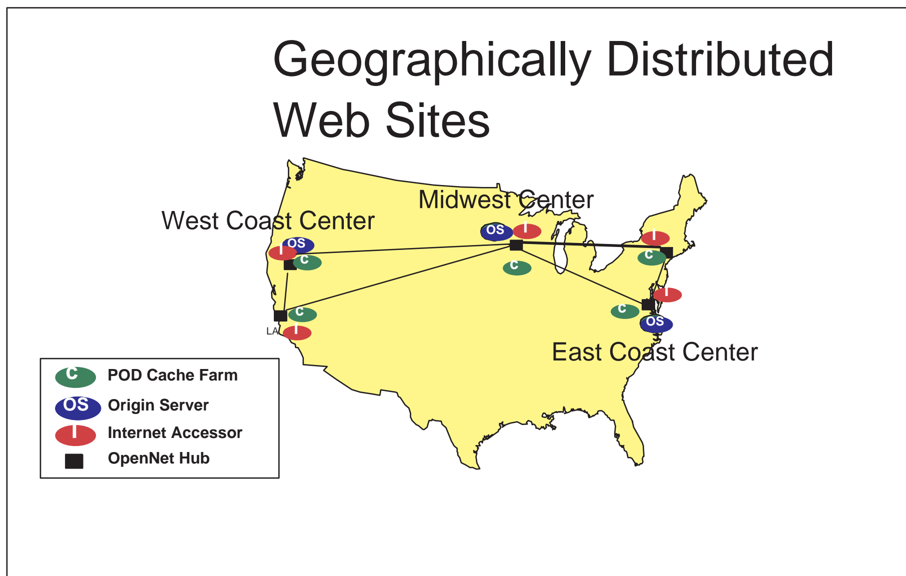

Figure 3: Geographically distributed Web sites are used by our system.

The Sport and Event Web sites use a multitier architecture, as shown in Figure 4. 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 five seconds. More recent Sport and Event Web sites also have two tiers of caches. Origin caches are reverse proxies in proximity to the origin servers. POD (Point of Distribution) 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 close to 90% at the POD caches are typically achieved, as demonstrated by our quantitative performance analysis of the system architecture; refer to Section 5. Our analysis further shows that, of the requests resulting in POD cache misses, about 50% result in origin cache hits. eNetDispatcher is a load balancer routing requests to the origin servers.

Figure 4 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

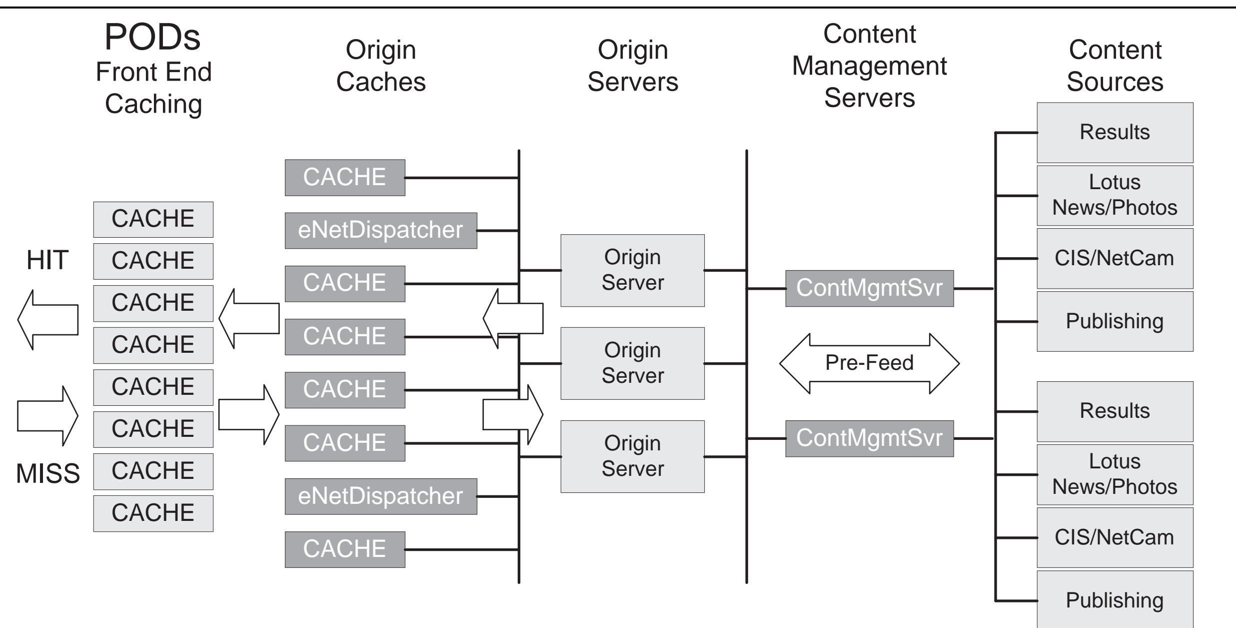

Figure 4: Multitiered serving architecture used by our system.

implementation makes it possible to decrease the cost of the installation by using commodity servers and caches.

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 are used together with our algorithms to control such decisions. Furthermore, as the request and update patterns for a page change, so too would these decisions regarding the setting of the expiration times for the page. 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.

The IBM Sport and Event Web sites have consistently been among the most popular Web sites in the world. The sites were recognized in the Guinness book of World Records in 1998. Two more recent statistics of significance are the following:

- 11.3 G *Total Event Hits* during the seventeen days of the 2000 Olympic Games.

- 1.4 M *Peak Hits Per Minute* requested during the Wimbledon 2001 Grand Slam Tennis Tournament.

4 Dynamic Data Request and Update Patterns

In this section we turn to present our analysis of the stochastic aspects of the client request patterns and the dynamic page update patterns exhibited at highly accessed Web sites providing significant dynamic content. As previously noted, the design of our system architecture and algorithms is based on such an analysis of the data from a few previous IBM Sport and Event Web sites, which includes Nagano [27]. A representative sample of some of the results of the corresponding analysis is therefore presented herein. We note, however, that our system architecture and algorithms have been successfully deployed for caching dynamic Web content at many IBM Sport and Event Web sites since Nagano, and the results presented in this (and the next) section are also representative of those found at these more recent Web sites. Furthermore, the stochastic models and methods briefly described in this section are used together with the results of our analysis as part of the DUP algorithms as discussed above. In addition to exploiting these results in our system design, the results of our analysis are of interest in their own right especially given the very limited previous research in the literature on dynamic page update patterns, as well as their interactions with the corresponding request patterns, in highly dynamic Web sites.

4.1 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 Origin servers in the Midwest site (see Figure 3). The number of requests and updates for dynamic pages were examined at different time scales. As a representative example, the graphs in Figure 5 illustrate the aggregate number of requests and updates encountered every 5 minutes at the East Coast and Tokyo sites. The request and update times have been adjusted to Greenwich Mean Time (GMT) in all the figures. 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 in 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 [9] for a different Web site environment. The update patterns similarly exhibit characteristics that persist over a broad range of time scales.

First consider 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 Figure 5 are primarily due to a strong public interest in Japan for events related to

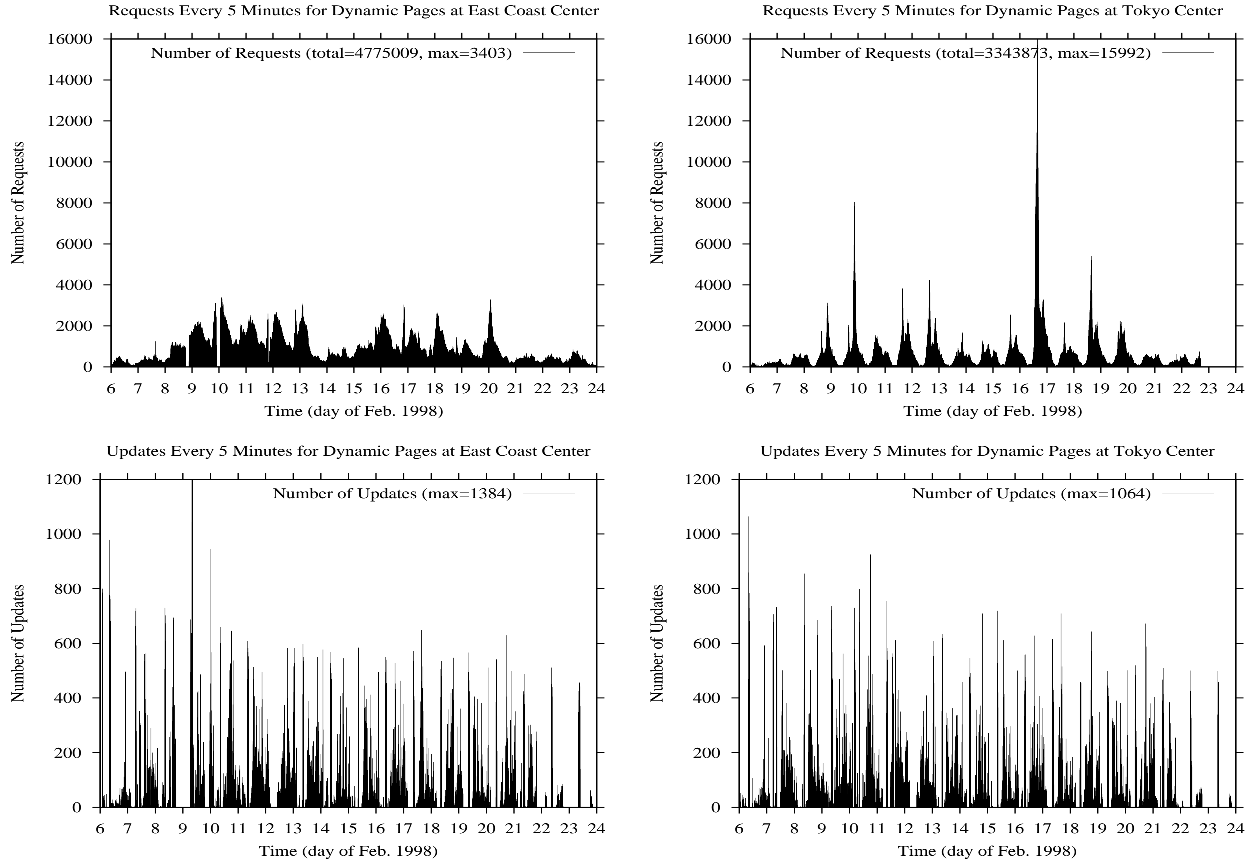

Figure 5: Request and update patterns every 5 minutes at the East Coast and Tokyo sites

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 they 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 is the limited bandwidth for the trans-Atlantic traffic, which may 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 trans-Atlantic 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 5.

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 pro-

cess 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 $\log \mathsf{P}[X > x] \sim -\alpha x^2$ for the East Coast site request data and $\log \mathsf{P}[X > x] \sim -\beta (\log x)^2$ for the Tokyo site request data, where $X \stackrel{\mathrm{d}}{=} X_n$ 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 the class of autoregressive processes to be quite suitable for our purposes. A (stationary) series of data points indexed by time, $\{Z_n\}$, is said to satisfy an order (p, q) *autoregressive moving average* process, denoted by ARMA(p, q), if it can be represented as $Z_n - \phi_1 Z_{n-1} - \ldots - \phi_p Z_{n-p} = \epsilon_n - \theta_1 \epsilon_{n-1} - \ldots - \theta_q \epsilon_{n-q}$, where p and q are respectively the orders of the AR and MA processes. A time series is an ARIMA(p, d, q) process if $(1 - B)^d Z_n$ is an ARMA(p, q) process, where $B Z_n = Z_{n-1}$ defines the backwards shift operator. Periodic, or seasonal, patterns can be additionally captured by extending this class of stochastic processes to the so-called *seasonal ARIMA* processes, denoted by 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 [5, 17] 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 non-stationary 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 [32, 33]. We therefore exploit these results in the algorithms and architecture presented in Sections 2 and 3.

It is important to note that in some Web site environments, such as many Sport and Event Web sites, a form of correlation exists among client requests as a result of an exogenous behavioral process. In the specific case of Nagano [27], 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 tends 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 Figure 5, 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 architecture described in Sections 2 and 3. 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 8th and 9th). These effects have been eliminated in more recent instances of our system design. Somewhat more minor differences in the two update processes illustrated in Figure 5 are due to small delays among the servers in receiving and processing updates. This is due in part to the geographically distributed system architecture.

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 $\log \mathsf{P}[U > u] \sim -\gamma u$ for the update data from both sites, where $U \stackrel{\mathrm{d}}{=} U_n$ 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 with some relatively strong periodic behavior. 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 [27]. In the interest of space, we provide in Figure 6 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 Figure 6, plots (a), (b), (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), (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 Zipf-like function. In particular, the function $f(x) = 200000/x$ provides a reasonably good fit, as illustrated in Figure 6(a). Figure 6(b) exhibits somewhat of a pattern where the more frequently requested pages turn out to be updated more frequently (with a number of exceptions). Figure 6(c) shows that the CPU overheads are relatively independent of the request frequencies. Similar to Figure 6(b), the request frequency curve indexed by the update rank in Figure 6(d) also exhibits a decreasing pattern, with a few exceptions. The update frequency curve indexed by the update rank in Figure 6(e) appears to be reasonably approximated by the function $f(x) = 4000/\sqrt{x}$, which is different from a Zipf-like function. It is interesting to note that the CPU overhead curve indexed by the update rank in Figure 6(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 has been and continues to be exploited to maintain good performance at the Sport and Event Web sites.

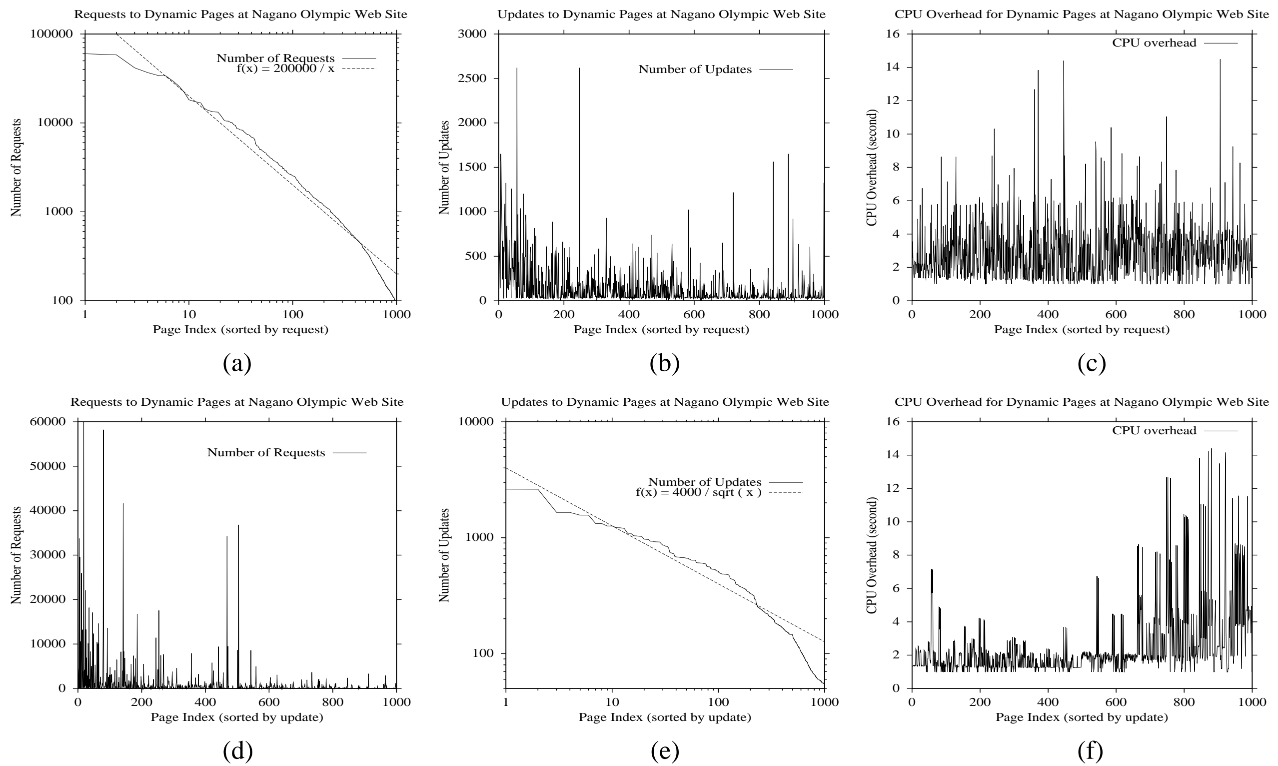

Figure 6: Request and Update Distributions to Dynamic Pages

4.2 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. Figure 7 plots the request and update processes for the 4 most frequently accessed pages from Europe. 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 minute, whereas requests for these same pages tended to arrive at rates of up to several thousand per second during peak periods. Further notice that both processes are nonstationary as they exhibit fairly strong periodic (daily) patterns. We also observe that the requests to certain pages (e.g., the 4th page) turn out to be fairly regular while the requests to some other pages (e.g., the 2nd page) turn out to be more bursty. It is interesting to note that the 4th page does not have many updates, and thus caching such pages may provide considerable performance gains given their request patterns. We further observe a certain degree of positive correlations between the requests and updates to the same page. The correlation factors for the request and update processes of these four 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 2.

Figure 8 plots the request and update processes for the 4 most frequently accessed pages from Asia. Once again, both processes are nonstationary with fairly strong periodic (daily) patterns. We observe that the requests to the 1st 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 4th 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. The correlation factors for the request and update processes of these four pages are 0.13, 0.18, 0.09 and 0.02, respectively.

To further understand the request and update processes in terms of interrequest and interupdate time statistics, Figures 9 and 10 plot the empirical distributions for the interrequest and interupdate processes of the 4 most frequently accessed pages from Europe and Asia, respectively. A more detailed view around the origin is embedded within each plot for a clearer illustration of the curve fitting. 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. A Weibull density function has the form

$$f(x) = \begin{cases} \frac{\gamma}{\theta} x^{\gamma-1} e^{-x^\gamma/\theta}, & x > 0; \\ 0, & \text{otherwise}; \end{cases} \quad (1)$$

for positive parameters θ and γ. 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 stationary

Poisson process is probably not sufficiently accurate for these types of processes encountered at the IBM Sport and Event Web sites. It is also interesting to note that the interupdate time, or lifetime, distributions for certain pages are multi-modal. This phenomenon is most significant for the 2nd and 4th most popular page from Europe and the 3rd and 4th 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, Figure 11 plots the interupdate tail distributions (in log scale) 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. 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 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 of the Pareto distribution is given by $\overline{F}(t) = (\beta/(\beta + t))^{\alpha}$, for $t \geq 0$, $\beta > 0$ and $0 < \alpha < 2$.

5 Performance Analysis

We now present our analysis of the performance benefits of the algorithms and architecture presented in Sections 2 and 3. As previously noted, this system design is based on such an analysis using the data from a few previous IBM Sport and Event Web sites. A representative sample of some of the results of the corresponding analysis using data from Nagano [27] is presented herein, quantifying the significant performance improvements of our system design. We note, however, that our system architecture and algorithms have been successfully deployed for caching dynamic Web content at many IBM Sport and Event Web sites since Nagano, and the results provided in this section are also representative of those found at these more recent Web sites as explained below.

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 [27] 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 architecture of Section 3, we first consider the case where POD caches are not exploited and thus all requests go directly to the Origin servers (see Figure 4). Note that this was the system architecture employed at Nagano [27]. In Figure 12 we plot the mean response times (in log scale) as a function of the number of servers under the request patterns found at 3 representative SP2 frames from the East Coast and Tokyo sites. The leftmost graph represents the cases in which the algorithms and architecture of Sections 2 and 3 are exploited, whereas the rightmost graph represents the cases in which the dynamic pages are served via the FastCGI interface in the standard manner. 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 Figure 12 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 Sport and Event Web sites, the algorithms and architecture 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 complete architecture of Section 3 in which POD cache farms are exploited in front of each site; refer to Figure 4. Recall that the complete architecture of Section 3 was subsequently employed at IBM Sport and Event Web sites following Nagano [27]. In Figure 13 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 3 representative SP2 frames from the East Coast and Tokyo sites. Once again, the leftmost graphs represent the cases in which our algorithms and architecture 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 results presented

are for a POD cache of size 128MB. Note that the corresponding results for larger cache sizes are identical to those provided in Figure 13, as the content data set at Nagano was around 100MB.

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 [27], we also consider the architecture of Section 3 with POD cache farms under workloads having inflated data sets that cover the full range of content data set sizes found at Sport and Event Web sites. In particular, we performed simulation experiments under the request patterns of the 3 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 for an inflation factor of 20 and POD cache sizes of 128MB (upper row), 256MB (middle row) and 512MB (lower row) are plotted in Figure 14, as a function of the number of servers. Similarly, the corresponding set of cache hit ratios are provided in Figure 15. Again, the leftmost graphs represent the measures under our approach and the rightmost graphs represent the measures under the standard methods together with POD cache farms. In comparison with Figure 13, 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 128MB to 256MB, with a somewhat smaller improvement as the cache size is increased from 256MB to 512MB. An even smaller relative improvement in performance is observed as the cache size is increased from 512MB to 1GB (not shown).

While the POD cache farms provide significant performance gains under both approaches, we note that the algorithms and architecture presented in this paper yield considerable improvements in performance (recall the log scale) 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 [27]. 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 main-

tain 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 architecture serve dynamic content at the performance level of serving static content.

Finally, it is important to further note that our experience at Nagano [27] and all other subsequent Sport and Event Web sites hosted by IBM up to the present time is consistent with the results of the foregoing performance analysis. In particular, system measurements based on Nagano confirm that the algorithms and architecture 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 architecture of Section 3 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.

6 Related Work

Zhu and Yang have developed a system for caching dynamic Web content in which data dependencies are managed at a coarse level of granularity [36]. 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 [26] in which caches were organized by sport and invalidation was performed on a sport-by-sport basis [15]. We found that the use of fine grained dependencies for precise updates and invalidations improved cache hit ratios considerably over coarse grained dependencies. Zhu and Yang have also developed techniques for lazy invalidation and selective recomputing in order to improve the performance of their system. Holmedahl, Smith, and Yang developed a system for cooperatively caching the results of requests for dynamic content [14]. 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 [30] developed a protocol to allow individual content-generating applications to exploit query semantics and specify how their results should be cached and/or delivered.

Vahdat and Anderson [34] proposed a scheme for caching dynamic Web content in which a tool they developed known as TREC 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. For example, as the authors point out, their approach is not applicable to Web sites in which dynamic content result from database queries. Hence, Vahdat and Anderson's approach would not work for any of the IBM Sporting and Event sites that we have worked on.

Cao, Zhang, and Beach have proposed a scheme for the caching of dynamic documents at Web proxies [7]. Their method migrates parts of server processing to caching proxies using what they call cache applets, where a cache applet

is server-supplied code that is attached to URLs or collections of URLs. Cache applets would typically be written in a platform-independent language such as Java. When an object is requested from the proxy cache, the corresponding cache applet is invoked with the client request and other information as arguments. The cache applet then decides what the proxy will send back to the user, either giving the proxy a new document to send back to the user, allowing the proxy to use the cached copy, or instructing the proxy to send the request to the Web server.

Douglis, Haro, and Rabinovich have proposed an extension to HTML that they call HPP in order to support dynamic document caching [10]. Their approach separates static and dynamic portions of a resource. The static portion, which can be cached, contains macro-instructions for inserting dynamic information. The dynamic portion contains the bindings of macro-variables to strings specific to the given access. The bindings are obtained for every access, and a template is expanded by the client prior to rendering the document.

There have been a number of papers published on Web cache consistency. Cao and Liu [6] compared three different consistency approaches: adaptive time-to-live, polling every time, and server-driven invalidation. Their paper presents the trade-offs of the approaches and argues that strong consistency via invalidation can be maintained with little or no extra cost than the current weak consistency approaches. A number of researchers have proposed using leases for Web cache consistency [35, 11]. Using this approach, servers notify caches of updates to cached content. Leases bound the amount of time that a server must continue to notify a cache of updates. Leases were originally proposed for distributed file cache consistency [12]. Gwertzman and Seltzer studied different cache consistency methods and found that a weak cache consistency protocol used in the Alex ftp cache offers a good balance of consistency and performance [13]. Li and Cheriton studied multicast invalidation and delivery of popular, frequently updated objects to proxy caches. They developed a protocol for grouping objects into volumes, each of which maps to an IP multicast group [21]. Krishnamurthy and Wills studied piggyback cache invalidation techniques for proxy caches [18, 19]. Their approach uses requests sent from the proxy cache to the server to improve coherency. When a proxy cache has a reason to communicate with a server, it piggybacks a list of cached, but potentially stale, resources from that server for validation. In responding to a proxy cache, servers can piggyback resources that have changed since the last access by the cache. The cache can then invalidate cached entries from this list and possibly extend the lifetimes of objects not on this list.

Delta encoding is a solution to the related problem of minimizing the bytes transferred in order to update cached objects [24]. In delta encoding, a cached object is updated by sending the differences between the new version of the object and one or more objects already present in the cache instead of sending the entire new object.

Techniques used to handle the large volume of traffic at Netscape's Web site are described by Mosedale, Foss, and McCool [25]. Kwan, McGrath, and Reed [20] describe the architecture of NCSA's Web site which was one of the most popular in the world a few years ago. Neither of these papers describe special techniques for handling dynamic content.

There has been a fair amount of work in the analysis of Web traffic patterns at the server level. Crovella and

Bestavros present evidence that the distribution of Web document transmission times are self-similar, possibly due to the distribution of Web document sizes, and that the distribution of browser silent times (i.e., when they are not making requests) are heavy-tailed, possibly due to the influence of user "think time" [9]. The study of Arlitt and Williamson found that the sizes of requested documents have a heavy tail similar to a Pareto distribution with $\alpha \approx 1$, and that the overall interrequest times had heavier than exponential tails [4]. Arlitt and Jin have analyzed the traffic from the 1998 World Cup Web site [2]. The study of Liu, Niclausse and Jalpa-Villanueva investigates the statistical characteristics of HTTP requests for various Web sites at the session level [22]. Arlitt, Krishnamurthy and Rolia [3] analyzed five days of workload data from a large Web-based shopping system to assess scalability and support capacity planning exercises for the multitier system. Padmanabhan and Qiu [29] have analyzed the request and update patterns of dynamic pages at the MSNBC Web site. Our research complements these studies in part by providing an analysis of a different Web server environment in which the density of requests at the server was significantly higher than at any of the Web sites in these previous studies, with the exception of [2]. Our study also is one of the first to investigate the corresponding update process encountered at highly accessed Web sites serving dynamic content, together with [29]. There are, however, considerable differences between our results and those in [29], 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 these update patterns in the design of a system architecture and algorithms to reduce the overhead of serving dynamic content. Finally, in comparison with the world-wide soccer event in [2], the Olympic Games considered in this paper consists of multiple events in different sports that take place simultaneously.

7 Conclusion

In this paper we have presented techniques for efficiently serving dynamic content at highly accessed Web sites and a quantitative analysis of the design motivation and performance benefits of these techniques. To reduce the overhead for generating dynamic content, we have developed the Data Update Propagation (DUP) algorithms for keeping cached content 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. DUP maintains data dependence information between cached objects and the underlying data that affect their values in a graph. When the system becomes aware of a change to the underlying data, graph traversal algorithms are applied to determine which cached objects are affected by the change. Cached objects that are found to be highly obsolete, relative to requests for the page, are then either invalidated or updated.

DUP is one of a few critical components of a general multitier architecture that has been deployed for high-performance serving of dynamic content to many clients at several Web sites hosted by IBM. The design of this architecture and the DUP algorithms is based on a detailed analysis of the stochastic aspects of the client request patterns and the dynamic page update patterns at a few previous highly accessed Web sites. Our considerable experience with this system design at subsequent highly accessed Web sites has proven to be consistent with the results of this

motivating analysis. In particular, our system design is able to achieve cache hit ratios close to 100% compared with 80% for an earlier version of our system which did not use this architecture and algorithms. As a result of these high cache hit ratios, Web sites based on our system design are able to serve content quickly even during peak request periods. High availability is achieved via redundant hardware and intelligent routing techniques. Our system architecture and algorithms for efficiently serving dynamic content provide more than an order of magnitude improvement in performance using an order of magnitude fewer servers over that obtained under existing methods for serving dynamic content. Performance benefits are realized by exploiting the wide range of stochastic properties of the request and update patterns of each page as demonstrated by the results of our analysis presented in this paper.

Much of the data in this paper was obtained by deployment of our system at sports 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. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35(R.1), HP Laboratories, Palo Alto, CA, September 1999.

[3] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the Scalability of a Large Web-Based Shopping System. *ACM Transactions on Internet Technology*, 1(1), August 2001.

[4] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 126–137, May 1996.

[5] Peter J. Brockwell and Richard A. Davis. *Time Series: Theory and Methods*. Springer-Verlag, 1987.

[6] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the 1997 IEEE International Conference on Distributed Computing Systems*, 1997.

[7] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of Middleware 1998*, 1998.

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

[9] Mark E. Crovella and Azer Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 160–169, May 1996.

[10] F. Douglis, A. Haro, and M. Rabinovich. HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[11] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of IEEE INFOCOM 2000*, 2000.

[12] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

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

[14] V. Holmedahl, B. Smith, and T. Yang. Cooperative Caching of Dynamic Content on a Distributed Web Server. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[15] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

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

[17] M.G. Kendall and J.K. Ord. *Time Series*. Oxford University Press, 1990.

[18] B. Krishnamurthy and C. Wills. Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[19] B. Krishnamurthy and C. Wills. Piggyback Server Invalidation for Proxy Cache Coherence. In *Proceedings of the 7th International World Wide Web Conference*, 1998.

[20] T. T. Kwan, R. E. McGrath, and D. A. Reed. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer*, 28(11):68–74, November 1995.

[21] D. Li and D. Cheriton. Scalable Web Caching of Frequently Updated Objects using Reliable Multicast. In *Proceedings of USITS '99*, 1999.

[22] Zhen Liu, Nicolas Niclausse, and Cesar Jalpa-Villanueva. Traffic model and performance evaluation of Web servers. *Performance Evaluation*, 46:77–100, October 2001.

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

[24] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of SIGCOMM '97*, 1997.

[25] D. Mosedale, W. Foss, and R. McCool. Lessons Learned Administering Netscape's Internet Site. *IEEE Internet Computing*, 1(2):28–35, March/April 1997.

[26] 1996 Atlanta Olympic Games Web Site, July 19 – August 4 1996. http://www.atlanta.olympic.org.

[27] 1998 Nagano Olympic Games Web Site, February 7 – February 22 1998. http://www.nagano.olympic.org.

[28] 2000 Sydney Olympic Games Web Site, September 15 – October 1 2000. http://www.olympics.com.

[29] Venkata N. Padmanabhan and Lili Qiu. The content and access dynamics of a busy web site: Findings and implications. In *Proceedings of ACM SIGCOMM '00*, pages 111–123, 2000.

[30] B. Smith, A. Acharya, T. Yang, and H. Zhu. Exploiting Result Equivalence in Caching Dynamic Web Content. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[31] J. Song, A. Iyengar, E. Levy-Abegnoli, and D. Dias. Architecture of a Web Server Accelerator. *To appear in Computer Networks*, 2002.

[32] Mark S. Squillante, David D. Yao, and Li Zhang. Web Traffic Modeling and Server Performance Analysis. In *Proceedings of the 38th IEEE Conference on Decision and Control (CDC)*, pages 4432–4439, 1999.

[33] Mark S. Squillante, David D. Yao, and Li Zhang. Internet Traffic: Periodicity, Tail Behavior, and Performance Implications. In Erol Gelenbe, editor, *System Performance Evaluation: Methodologies and Applications*. CRC Press, 2000.

[34] A. Vahdat and T. Anderson. Transparent Result Caching. In *Proceedings of the 1998 Annual USENIX Technical Conference*, June 1998.

[35] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, 1998.

[36] H. Zhu and T. Yang. Class-based Cache Management for Dynamic Web Content. In *Proceedings of IEEE INFOCOM 2001*, April 2001.

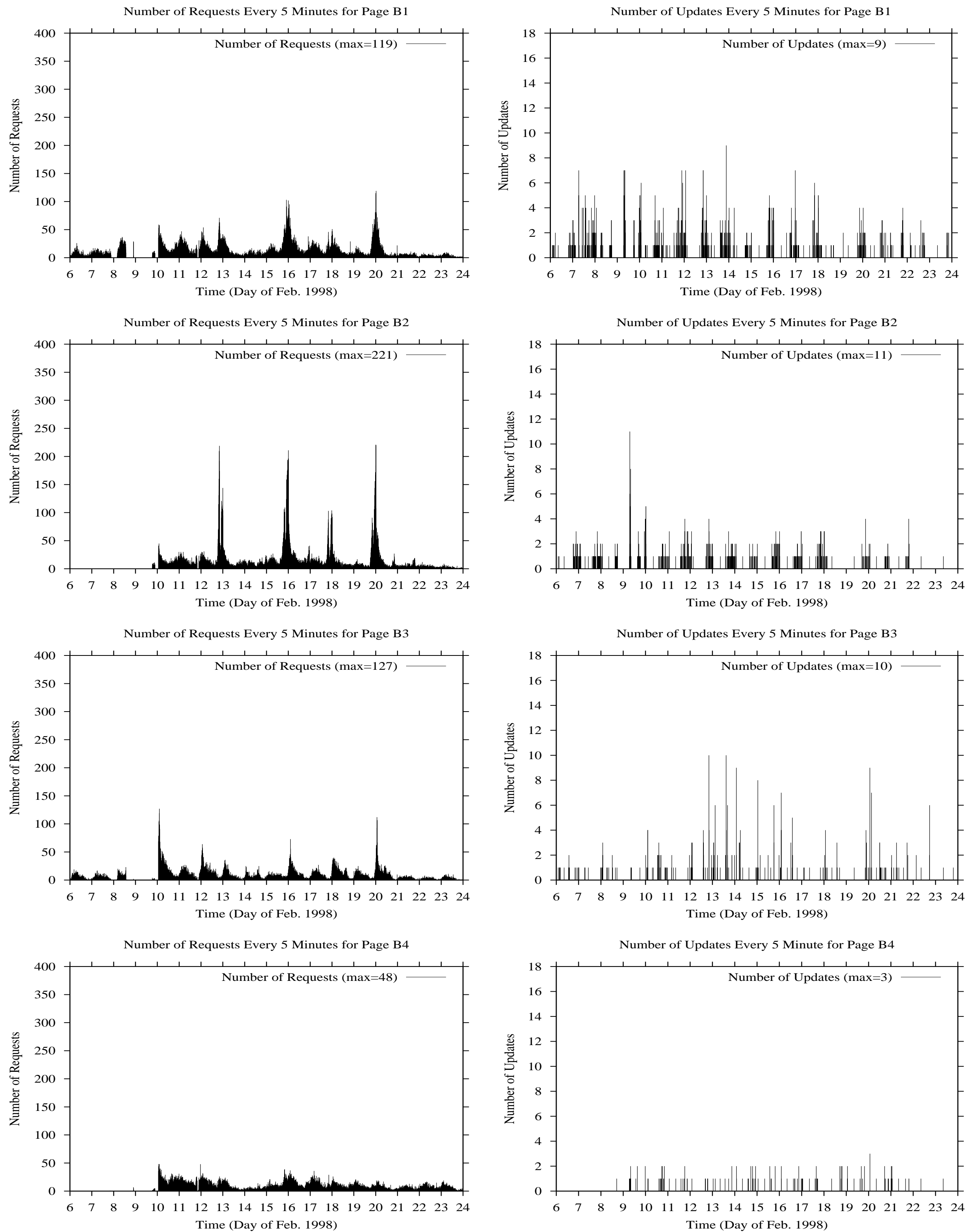

Figure 7: Requests and updates for the 4 most popular dynamic pages from Europe

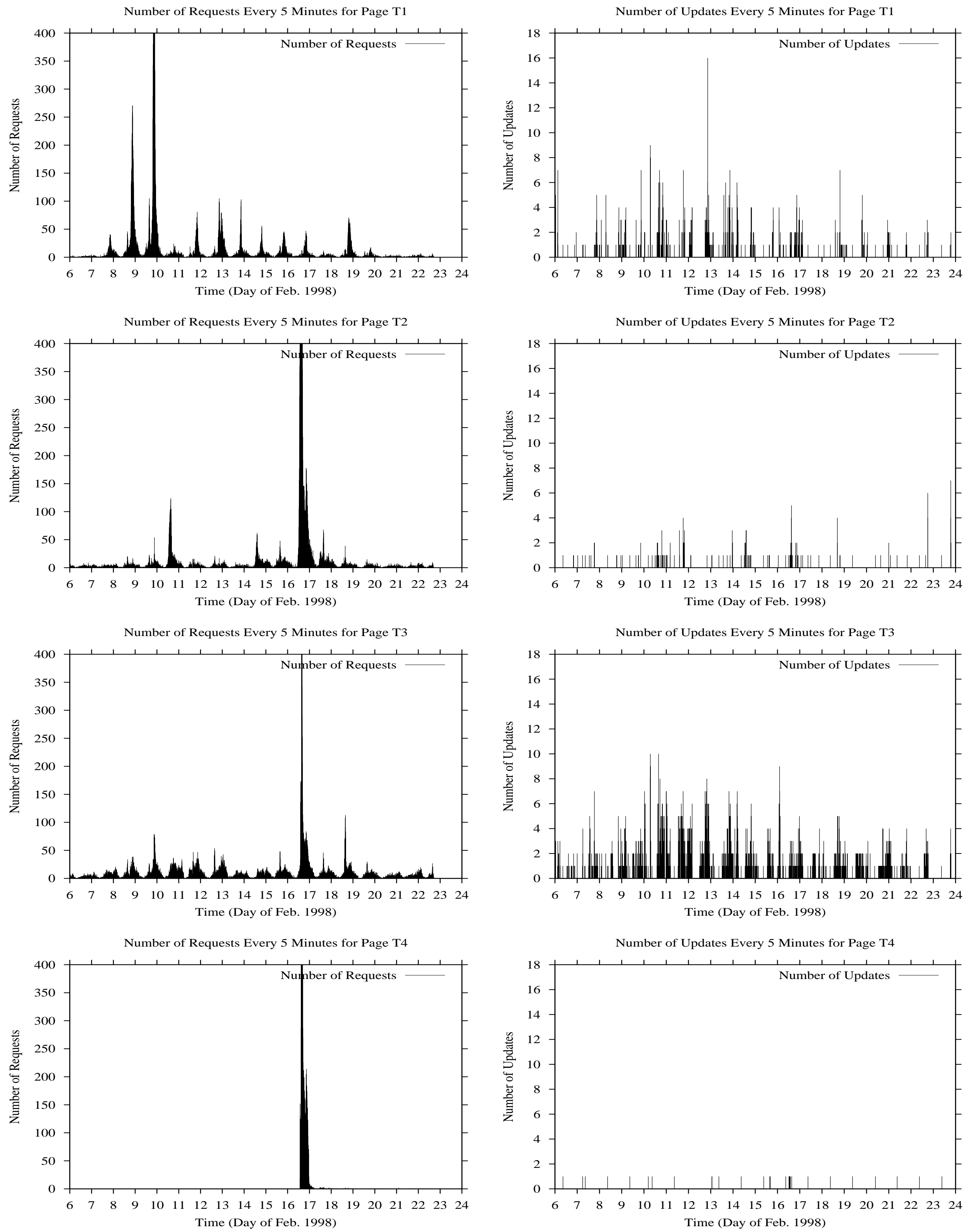

Figure 8: Requests and updates to the 4 most popular dynamic pages from Asia

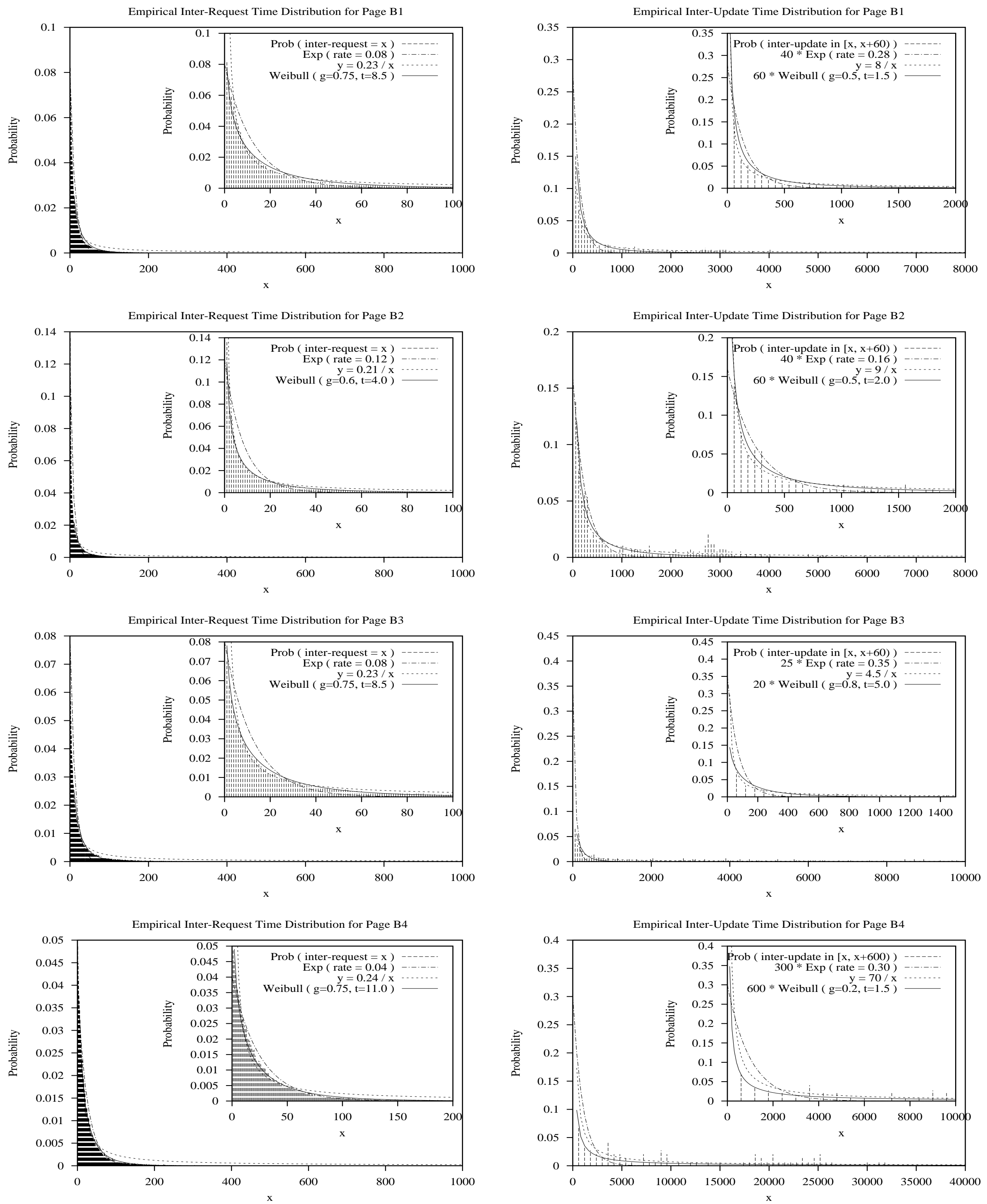

Figure 9: Interrequest and Interupdates times for the 4 most popular dynamic pages from Europe

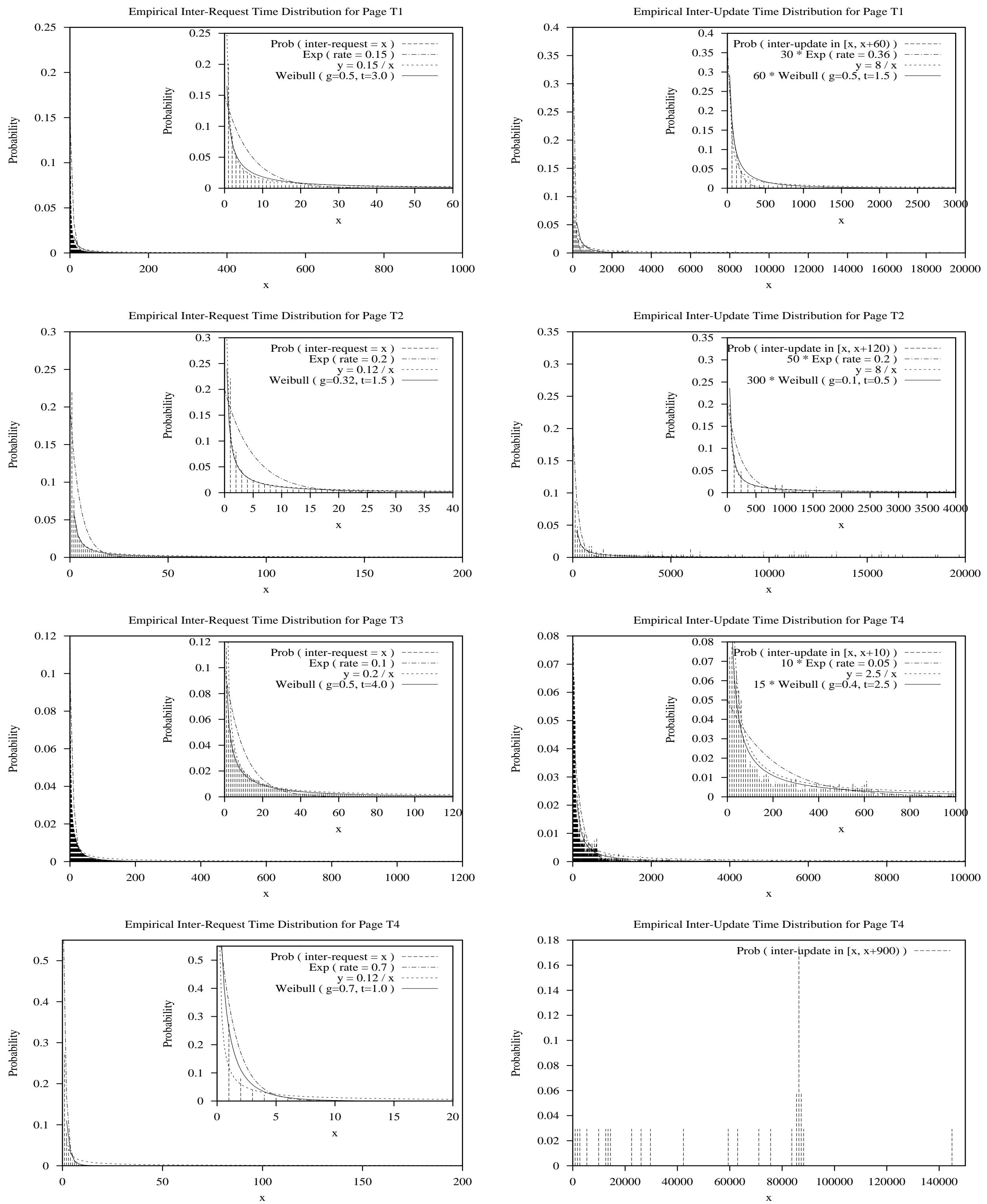

Figure 10: Interrequest and Interupdates times for the 4 most popular dynamic pages from Asia

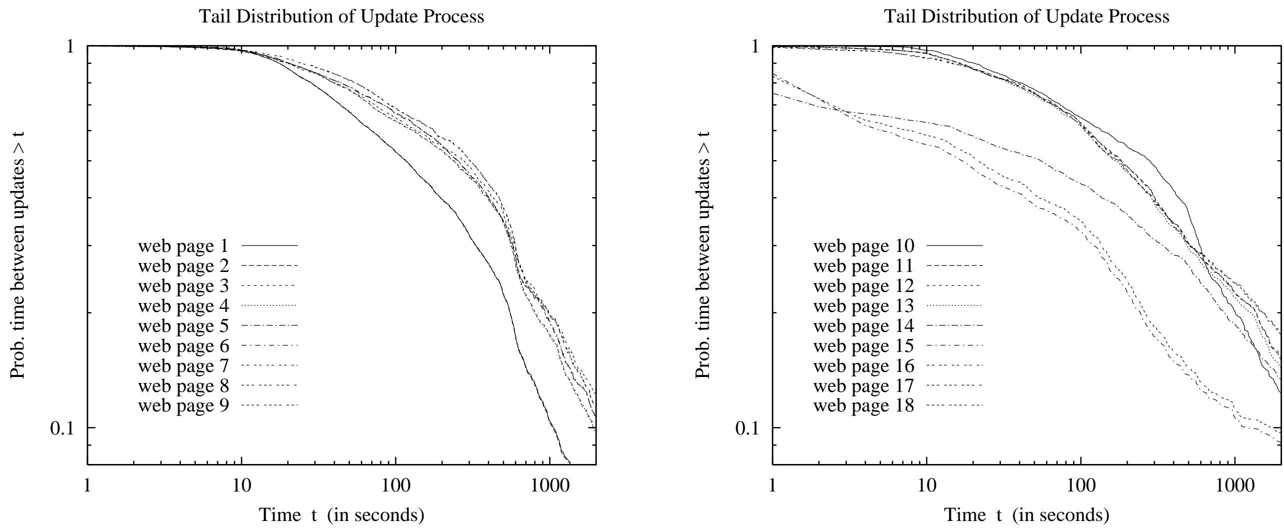

Figure 11: Tail Distributions of update processes for a representative set of individual dynamic pages.

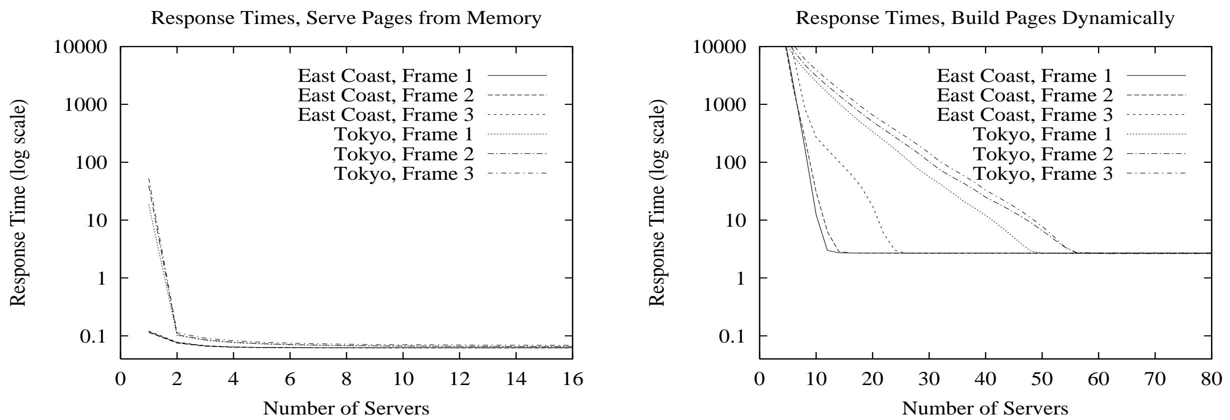

Figure 12: Mean response times for 3 representative SP2 frames from the East Coast and Tokyo sites.

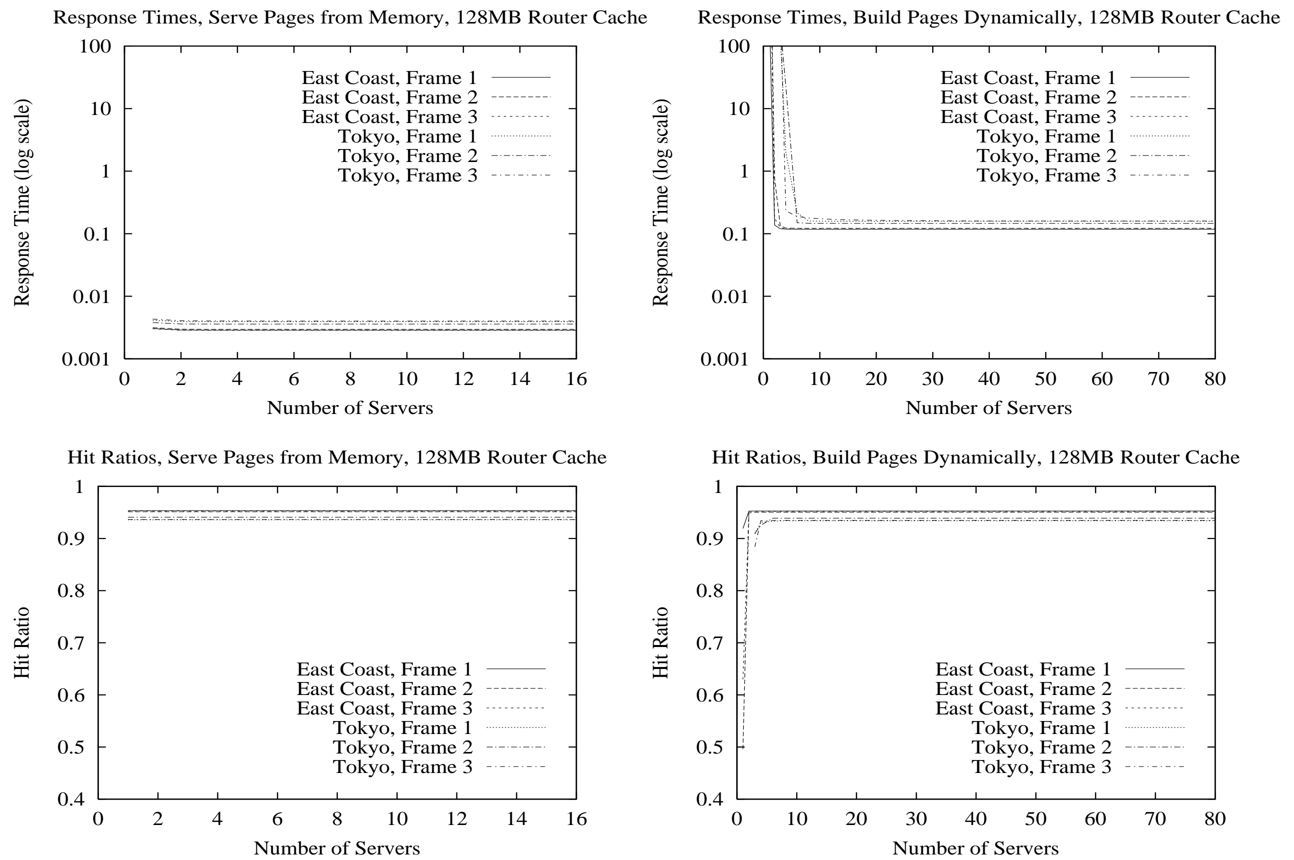

Figure 13: Mean response times and cache hit ratios for 3 representative SP2 frames from the East Coast and Tokyo sites, each with a POD cache farm.

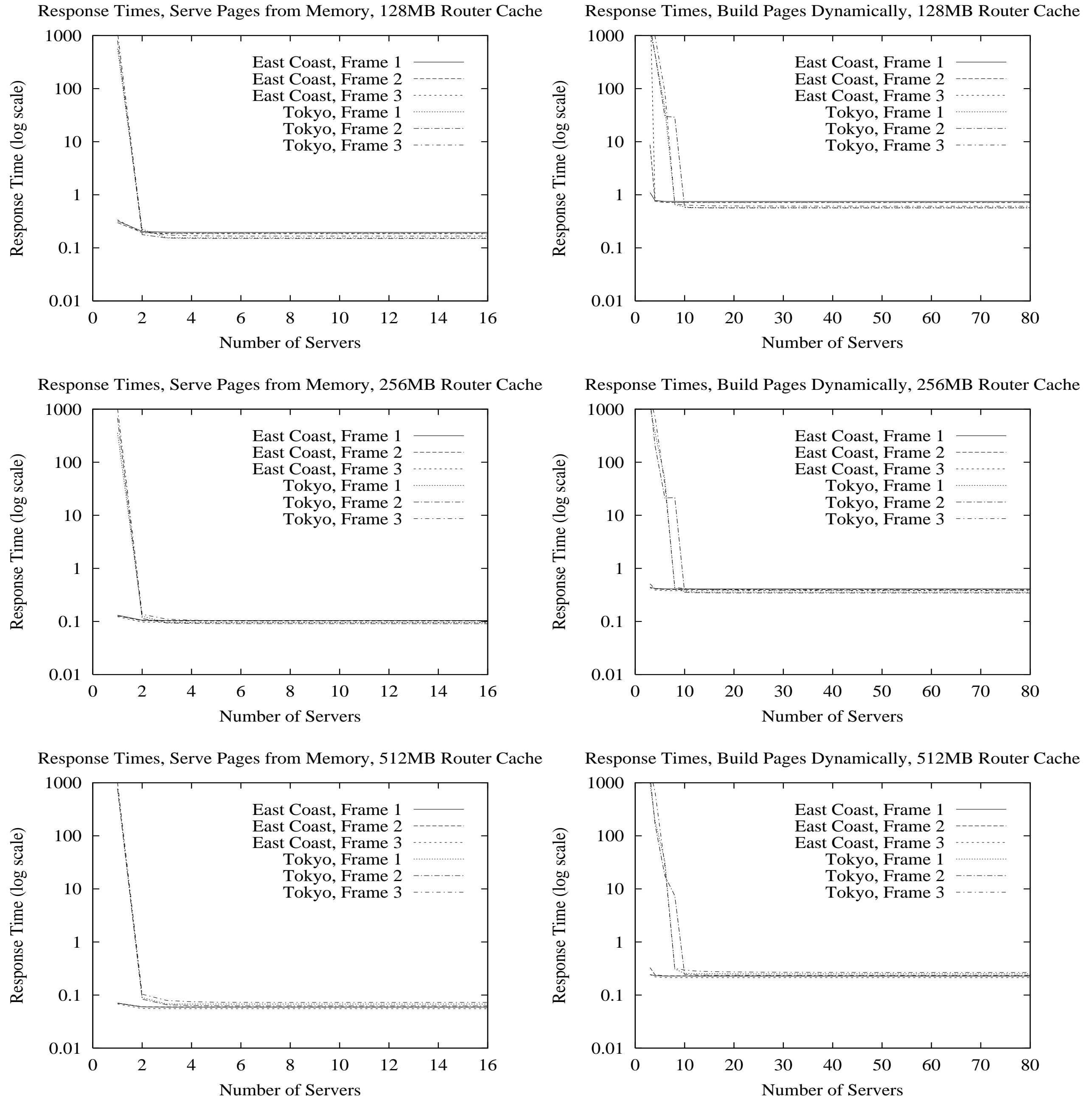

Figure 14: Mean response times for 3 representative SP2 frames from the East Coast and Tokyo sites, each with a POD cache farm, under an inflated Nagano data set.

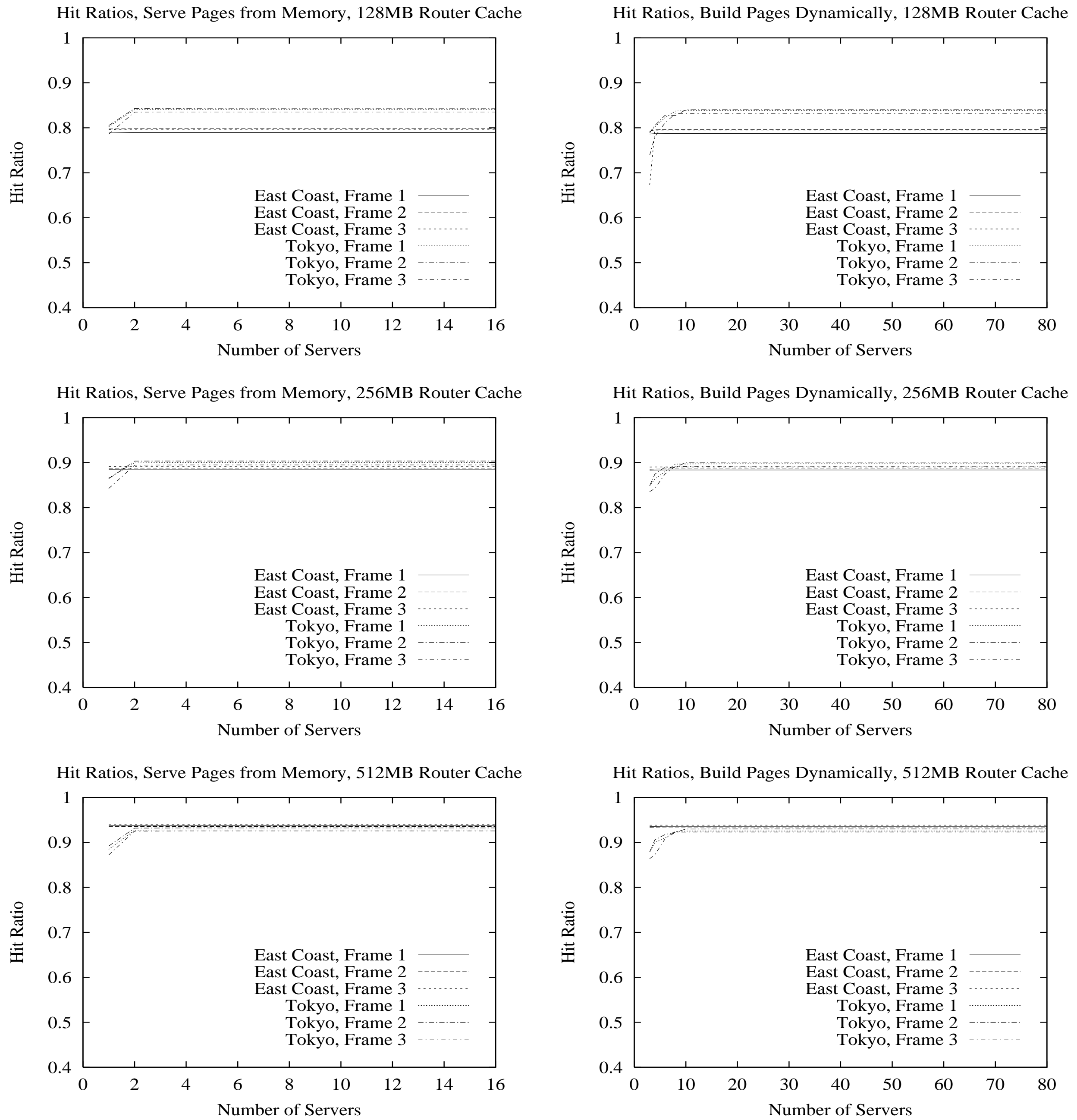

Figure 15: Cache hit ratios for 3 representative SP2 frames from the East Coast and Tokyo sites, each with a POD cache farm, under an inflated Nagano data set.