A Scalable System for Consistently Caching Dynamic Web Data

Jim Challenger, Arun Iyengar, and Paul Dantzig
IBM Research
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

Abstract

This paper presents a new approach for consistently caching dynamic Web data in order to improve performance. Our algorithm, which we call Data Update Propagation (DUP), maintains data dependence information between cached objects and the underlying data which affect their values in a graph. When the system becomes aware of a change to underlying data, graph traversal algorithms are applied to determine which cached objects are affected by the change. Cached objects which are found to be highly obsolete are then either invalidated or updated. DUP was a critical component at the official Web site for the 1998 Olympic Winter Games. By using DUP, we were able to achieve cache hit rates close to 100% compared with 80% for an earlier version of our system which did not employ DUP. As a result of the high cache hit rates, the Olympic Games Web site was able to serve data quickly even during peak request periods.

1 Introduction

Web servers provide two types of data: static data from files stored at a server and dynamic data which are constructed by programs that execute at the time a request is made. Dynamic pages can seriously reduce Web server performance. High-performance Web servers can typically deliver up to several hundred static files per second on a uniprocessor. By contrast, the rate at which dynamic pages are delivered is often orders of magnitude slower; 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 [6, 7].

Dynamic pages are essential at Web sites which provide data that change frequently. If pages are generated dynamically by a server program, the server program can return the most recent version of the data. If, on the other hand, files are created to serve the pages statically, it may not be feasible to keep the files current. This is particularly true if the number of files which need to be updated frequently is large. Consequently, the official Web site for the 1998 Olympic Winter Games generated a high percentage of Web pages dynamically [3]. Whenever new content became available to the computers implementing the Web site, updated Web pages reflecting these changes were made available to the rest of the world within seconds. Clients could thus rely on the Web site to provide the latest results, news, photographs, and other information from the Olympic Games.

Since the Olympic Games Web site was one of the most popular in the world for the duration of the Olympic Winter Games, performance was critical. One of the most important techniques we used for improving performance at the Olympic Games Web site was to cache dynamic pages the first time they were created. That way, subsequent requests for the same dynamic page could access the page from a cache instead of repeatedly invoking a program to generate the same page.

A key problem with caching dynamic pages is determining what pages should be cached and when a cached page has become obsolete. Our cache provides API's which allow an application program to explicitly add, delete, and update cached objects [4]. Explicit management of the cache is essential for optimal performance and consistency. However, API's for explicitly managing the contents of caches are not sufficient for achieving both good performance and cache consistency. At the official Web site for the 1996 Olympic Summer Games, an earlier version of our cache was used which also had API's for explicitly adding, deleting, and updating objects. One of the problems we encountered during the 1996 Olympic Games 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 were invalidated, many current pages were also invalidated. This caused high miss rates after the system received new information.

Using our experience from the 1996 Olympic Games, we developed a new algorithm called *Data Update Propagation (DUP)* for precisely identifying which cached pages have become obsolete as a result of new information received by the system. DUP significantly reduced the number of cached pages which needed to be invalidated or updated after new information was received. By using DUP in combination with prefetching, the 1998 Olympic Games Web site achieved cache hit rates of close to 100% compared to around 80% for the 1996 Olympic Games Web site. The high cache hit rates allowed the system to serve pages quickly even during peak request periods.

2 The Data Update Propagation Algorithm

Data update propagation (DUP) determines how cached Web pages are affected by changes to underlying data which determine the current values 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 values have not changed may be mistakenly invalidated or updated from a cache after a database change. Such unnecessary updates to caches can increase miss rates and hurt performance.

DUP maintains correspondences between *objects* which are defined as items which may be cached and *underlying data* which periodically change and affect the values of objects. Although an entity may be both an object as well as underlying data, objects and underlying data could also be different, which would mean that underlying data are not cacheable. In our system, caches may contain both entire HTML pages and fragments of HTML pages. It is possible for a cached HTML fragment f to affect the value of a cached HTML page. In this situation, f would constitute both an object and underlying data. In a simpler system, caches may consist entirely of HTML pages and underlying data may consist entirely of parts of databases. In this case, the underlying data and objects would be disjoint.

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 which it has stored in order to determine which cached objects are affected. Caches use dependency information to determine which objects need to be invalidated or updated as a result of changes to underlying data.

Our cache architecture centers around a *cache manager* which is a long-running daemon process managing storage for one or more caches. 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. 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 which is 5 times the weight of the latter edge.

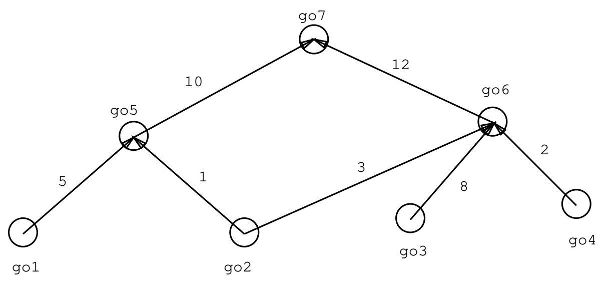

Figure 1: An object dependence graph (ODG). Weights are correlated with the importance of data dependencies.

In many 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 ODG's. We then show how DUP can be generalized to any ODG.

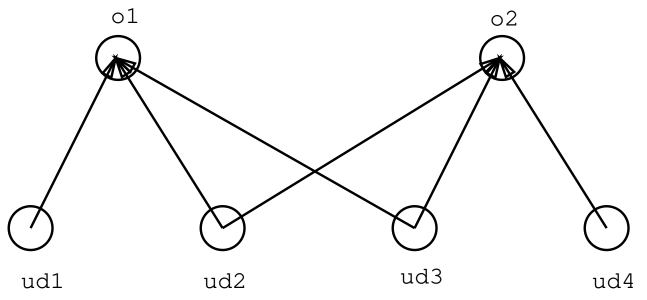

Figure 2: A simple object dependence graph.

2.1 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 which 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 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 $o2$ to be invalidated. The function call:

$$underlying\_data\_has\_changed(ud2)$$

would cause both $o1$ and $o2$ to be invalidated.

Cache managers maintain cache directories containing information about cached objects. Directory information for a cached object with one or more dependencies on underlying data includes the object ID and an *incoming adjacency list* containing all underlying data ID's corresponding to underlying data which affect the value of the object. Figure 3 depicts the incoming adjacency lists corresponding to the graph in Figure 2.

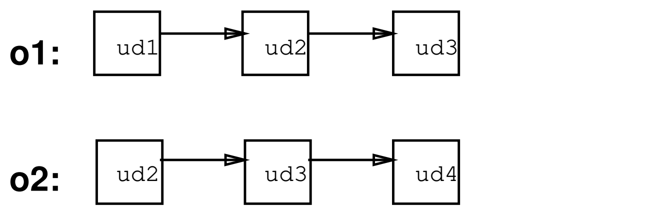

Figure 3: Incoming adjacency lists corresponding to the graph in Figure 2.

Cache managers also maintain hash tables containing pointers to *outgoing adjacency lists* for nodes in *G* corresponding to underlying data. Hash tables are indexed by underlying data ID's. Each outgoing adjacency list contains the object ID's of objects whose values depend on the underlying data represented by the underlying data ID. Figure 4 depicts the hash table corresponding to the graph in Figure 2.

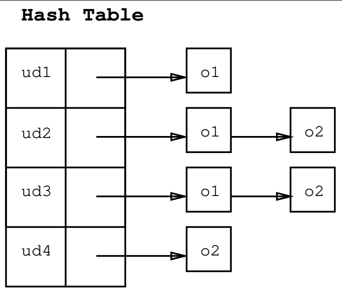

Figure 4: The hash table corresponding to the graph in Figure 2.

An invocation of the API function:

$$add\_dependency(obj\_id, ud\_id)$$

adds a new edge to *G* by adding obj_id to the outgoing adjacency list for ud_id and ud_id to the incoming adjacency list for obj_id.

An invocation of the API function:

$$underlying\_data\_has\_changed(ud\_id)$$

is implemented by invalidating all objects on the outgoing adjacency list for ud_id.

It is sometimes desirable to delete nodes from *G*. For example, all dependencies on an underlying data node could become obsolete in which case the node would no longer be needed. Similarly, an object could go away which would make its node in *G* unnecessary. Object nodes are removed from *G* by removing the object ID from outgoing adjacency lists for all nodes on the incoming adjacency list for the object and removing the incoming adjacency list for the object. Underlying data nodes are removed from *G* by removing the underlying data ID from incoming adjacency lists for all nodes on the outgoing adjacency list for the underlying data node and removing the hash table entry for the underlying data node.

2.2 Generalizing DUP to Arbitrary Object Dependence Graphs

2.2.1 Overview

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

- The generalized DUP algorithm is 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.

  It is not necessary for a node to correspond to an object or underlying data. A node which does not correspond to an object or to underlying data is said to represent a *virtual object*. The purpose of a virtual object is usually to propagate change information to nodes representing real objects.

- In some cases, it is acceptable for cached objects to be slightly out of date. Retaining out of date objects in a cache can be cheaper than always updating or invalidating objects after they 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 algorithm provides a metric for determining how obsolete a cached object is. In order to enable the metric, edges of the ODG should have weights which are correlated with the importance of dependencies such as in Figure 1.

- A cache manager might be managing multiple caches. In this situation, the generalized DUP algorithm allows a single ODG to be applied to multiple caches. Different caches may concurrently be storing different versions of the same object.

- The generalized DUP algorithm provides a metric for quantitatively assessing how similar two versions of the same object are. This is particularly useful when different versions of the same object are being stored in different caches.

When an application program notifies a cache manager that underlying data has changed, the cache manager identifies all objects which 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. 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, two objects are consistent with each other if either:

1. Both of them are current.

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

2.2.2 Data Maintained by the Generalized DUP Algorithm

Each vertex of G has an ID field representing it. For simplicity, we will assume that object nodes, underlying data nodes, and virtual graph object nodes share the same ID space. The cache manager stores information about each vertex in G in a *vertex information block (VIB)*. The cache manager also maintains a hash table which is indexed by vertex ID's and contains pointers to VIB's. That way, the cache manager can locate the VIB corresponding to a vertex ID in constant time.

A VIB for a node $n1$ has the following fields:

- ID : A string identifying the object, underlying data, or virtual object corresponding to $n1$.
- $update\_num$: The number of updates the cache manager is aware of which have been performed on $n1$. The cache manager assigns version numbers to different version numbers of the same object. The version number for a particular version of an object is the $update\_num$ field at a time the version of the object was current.
- $timestamp$: Represents the time of the last change to $n1$ which the cache manager is aware of. 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.
- $cache\_list$: If $n1$ corresponds to an object, a list identifying all caches containing the object.
- $incoming\_dep$: The incoming adjacency list for $n1$. Each element of $incoming\_dep$ corresponds to an edge terminating in $n1$ and contains two components:
  1. The ID of the node which is the source of the edge.
  2. The weight of the edge.
- $outgoing\_dep$: The outgoing adjacency list for $n1$. It consists of ID fields for each node $n2$ for which a vertex from $n1$ terminating in $n2$ exists.
- sum_weight : The sum of the weights of all edges terminating in $n1$.
- $threshold\_weight$: If $n1$ corresponds to an object $o1$, 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 $n1$ 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.
- $latest\_object$: If $n1$ corresponds to an object, a pointer to the latest cached version of the object which the cache manager is aware of. Caches use this pointer to obtain recent versions of objects from other caches. This avoids the overhead of having to calculate the objects from scratch.

The structure of G is stored in the ID, $incoming\_dep$, $outgoing\_dep$, sum_weight, and $threshold\_weight$ fields in VIB's. Application programs specify the structure of G via API functions which modify these fields.

Each cached version of an object has an *object information block (OIB)* associated with it. Each cache stores OIB's for all objects contained in the cache in a directory. Pointers to OIB's are maintained in a hash table indexed by object ID's. That way, the cache manager can locate an OIB corresponding to an object in a specific cache in constant time.

An OIB for an object $o1$ has the following fields:

- ID : A string identifying $o1$.
- $version\_num$: Different versions of the same object have different $version\_num$ fields. The $version\_num$ field for a particular version of $o1$ is equal to the $update\_num$ field of the corresponding VIB for $o1$ at a time when the version of $o1$ was current.
- $timestamp$: represents the time at which the cached version of $o1$ became current.
- $actual\_sum\_weight$: 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 $n2$.
- dep_list : The incoming adjacency list for $o1$. Each element of dep_list corresponds to an edge from a node in the graph $n2$ terminating in the node corresponding to $o1$ and contains the following components:
  1. The ID for $n2$.
  2. $weight\_act$: A number representing how consistent the current version of $n2$ is with the cached version of $o1$. Our algorithm uses values of 0 (totally inconsistent) or the weight of the corresponding edge in the graph (totally consistent). A straightforward extension is to allow values in between these two extremes to represent degrees of inconsistency.
  3. $consistent\_version\_num$: the $update\_num$ field in the VIB for $n2$ at the time the cached version of $o1$ was current.

2.2.3 Adding Objects to Caches

When the current version of an object $o1$ is added to a cache $c1$, the cache manager ensures that an OIB for $o1$ exists in the directory for $c1$. This may require creating a new OIB for $o1$. The $version\_num$, $timestamp$, and $actual\_sum\_weight$ fields of the OIB are set to the $update\_num$, $timestamp$, and sum_weight fields respectively of the corresponding VIB fields. The dep_list field in the OIB is copied from the $incoming\_dep$ field in the VIB. For each node $n2$ on the dep_list for $o1$, the $consistent\_version\_num$ field is set to the $update\_num$ field in the VIB for $n2$. A pointer to $c1$ is added to the $cache\_list$ field of the VIB for $o1$ if $o1$ was not previously contained in $c1$.

Noncurrent versions of objects may be copied from one cache to another. If so, the OIB for the object is also copied to the new cache.

2.2.4 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 $update\_num$ field in the VIB for $n1$ by 1. The $timestamp$ field in the VIB is set to $num\_updates$. This indicates that $n1$ has been visited during the current graph traversal. If $n1$ corresponds to an object, the $cache\_list$ field in the VIB is traversed in order to notify 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 object 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 for $n2$ is equal to $num\_updates$. If $n2$ has not been visited yet, its $update\_num$ field is incremented by 1, and its $timestamp$ field is set to $num\_updates$. If $n2$ corresponds to an object $o2$, all caches containing $o2$ must update the OIB for $o2$ and possibly update or invalidate their copies of $o2$. Such caches are located by traversing the $cache\_list$ field in the VIB for $n2$. For each such OIB, the $actual\_sum\_weight$ field is decremented by the $weight\_act$ field in the OIB corresponding to the edge $(n1, n2)$. If this results in the $actual\_sum\_weight$ field of the OIB being less than the $threshold\_weight$ field in the VIB, $o2$ is either invalidated from the cache or replaced with a more recent version.

If $actual\_sum\_weight >= threshold\_weight$, $o2$ is not replaced with a new version. Instead, the $weight\_act$ field in the OIB corresponding to the edge $(n1, n2)$ is set to 0.

If, on the other hand, $n2$ has already been visited, a slightly different procedure is followed. If $n2$ corresponds to an object $o2$, all caches containing $o2$ must update the OIB 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 OIB with the $update\_num$ field in the VIB. If the cached version of $o2$ is current, the $consistent\_version\_num$ component in the dep_list element corresponding to the edge $(n1, n2)$ is set to the $update\_num$ field for $n1$.

If, on the other hand, the cached version of $o2$ is not current, the $actual\_sum\_weight$ field is decremented by the $weight\_act$ field in the OIB corresponding to the edge $(n1, n2)$. If this results in the $actual\_sum\_weight$ field of the OIB being less than the $threshold\_weight$ field in the VIB, $o2$ is either invalidated from the cache or replaced with a more recent version.

If $actual\_sum\_weight >= threshold\_weight$, $o2$ is not replaced with a new version. Instead, the $weight\_act$ field in the OIB corresponding to the edge $(n1, n2)$ is set to 0.

2.2.5 Comparing Two Versions of the Same Object

Our algorithm provides a similarity score for assessing how similar two versions of the same object $o1_1$ and $o1_2$ are. Let $n1$ be the node in G corresponding to $o1$. The similarity score is based on the sum of the weights of incoming dependencies to $n1$ from nodes $n2$ for which the same version of $n2$ is consistent with both $o1_1$ and $o1_2$. Let $common\_weight$ be this sum. The similarity score is then given by the formula:

$$SS = \frac{common\_weight}{sum\_weight}$$

where sum_weight is obtained from the VIB for $n2$. Similarity scores range from 0 (least similar) to 1 (most similar).

In order to calculate $common\_weight$, the cache manager adds up the sum of the weights of all edges $(n2, n1)$ in G such that the $consistent\_version\_num$ component corresponding to the edge are the same in the OIB's for both $o1_1$ and $o1_2$.

3 DUP Implementation at the 1998 Olympic Games

3.1 Prefetching Pages

One of the key techniques we used to obtain cache hit rates close to 100% was to calculate and cache new versions of pages immediately after it was determined that the pages were obsolete instead of invalidating the pages and waiting for them to be loaded on demand. Consequently, once a frequently requested page was cached, a request for the page would always result in a cache hit.

This technique was effective because frequently requested pages were requested far more frequently than they were updated. For example, pages for sports which were in progress changed as often as once or twice per minute. However, requests for the "sports-in-progress" pages tended to arrive at rates of up to several thousand per second during peak periods.

During the 1996 Summer Olympic Games, 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.

For the 1998 Winter Olympic Games in Nagano, when data changed, we used Data Update Propagation 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.

3.2 Dynamic Compound Pages

A dynamic compound page is a page which consists of multiple fragments of information, any of which may change independently over time (Figure 5). Any fragment may itself be composed of smaller fragments, to an arbitrary level of nesting.

A change to the underlying data from which a dynamic compound page has been composed is likely to affect only some of the fragments comprising the page.

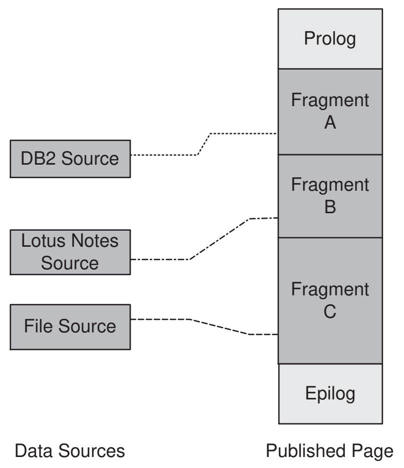

Figure 5: Dynamic compound pages.

Our system caches fragments of dynamic compound pages and constructs dynamic compound pages from the underlying fragments. ODG's are used to represent dependencies between fragments and dynamic compound pages. The fragments needed to construct dynamic compound pages are obtained by traversing ODG's. Consequently, generation of multiple pages which are constructed from the same fragment will only result in the fragment being created once. By contrast, the conventional method of generating pages independently of each other results in the data comprising the fragment being recalculated for each page. If the fragment is expensive to generate, our method results in considerable savings over the conventional method.

3.3 Overview of the Trigger Monitor

It was discovered during implementation of the Olympic Games system that significant optimizations could be realized by maintaining only a portion of the ODG in the cache and dynamically calculating the subset of the ODG related to a specific change when that change occurred. Savings included:

- Space savings. Only a subset of the ODG physically existed at any given time.
- Reliability and complexity savings. The ODG is a rather complex structure. If a system fails, the ODG must be restored rapidly if that system is to be made functional again quickly. Since the most dynamic and complex portion of the ODG was calculable on demand, it was possible to implement an extremely fast, reliable, and simple automated backup and restore algorithm for the ODG.

The *trigger monitor* calculates portions of the ODG including the nodes corresponding to entire HTML pages. The trigger monitor is an event-driven process. When data changes (e.g. a row is inserted into a DB2 database, a photograph is published), the entity making the change notifies the trigger monitor of the change. When the trigger monitor is notified of changes to underlying data, it expands the relevant portion of the ODG, coordinates the building or rebuilding of relevant pages, and broadcasts new and updated pages to the serving nodes. Persistent state is maintained for each transaction reporting a data change from the time of acceptance to the time the transaction leaves the system to guarantee processing of the event in the face of system or component failure. Retry and commit logic is used to insure delivery of messages and proper composition of pages. Figure 6 shows how the trigger monitor fits in the overall system.

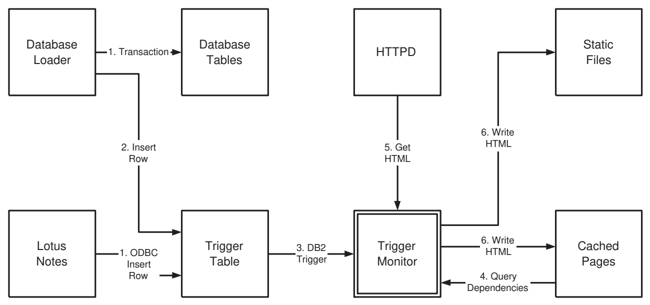

Figure 6: The system components used to provide updated pages.

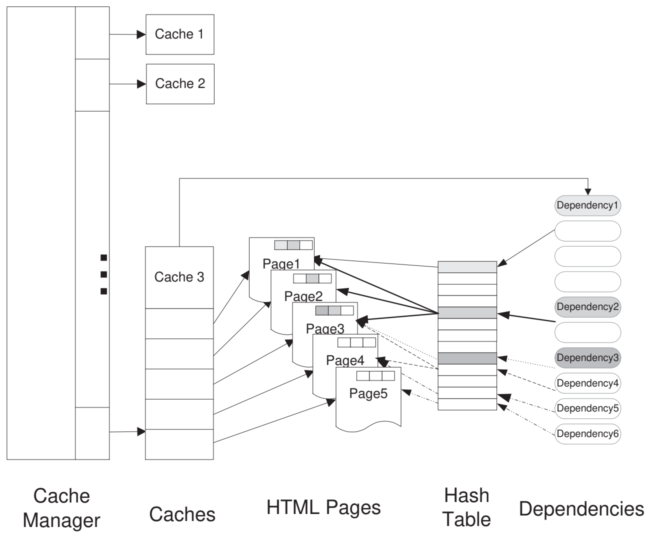

Figure 7: The stored portion of the ODG. These dependency-to-page relationships are stored in caches.

When a data change is detected, the object dependence graph (ODG) for the event must be traversed. Since only a portion of the ODG is explicitly stored in the system, we must generate the rest of the ODG as it pertains to the current event. When a page is composed and cached, the dependencies representing the data from which the page is built are also cached. Given a page, it is possible to find its dependencies; given a dependency, it is possible to find the pages dependent on it (Figure 7). Figure

8 is a representation of a portion of this relationship as used in the Olympic Games.

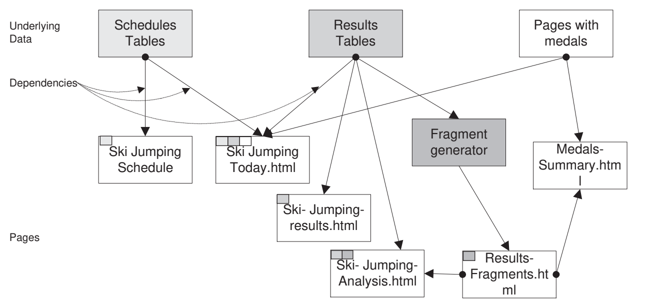

Figure 8: Another depiction of the stored portion of the ODG.

When an event occurs, for example, a skier wins, or a new event schedule arrives, a summary of that event is sent to the trigger monitor. The summary record is expanded into the set of cache dependencies corresponding to that event. This constitutes one of the calculations required to fully identify the ODG of an event. In Figure 9, the "trigger events (a)" occur as the result of data update events. From these events, it is possible to calculate the dependencies that comprise this portion of the ODG.

Some events do not have explicitly stored dependencies. Rather, it is possible to determine directly from the event code what pages are affected. An example was the arrival of a new sports photo. The data event this triggered was encoded so that the relevant URLs could be directly computed. This computation is another calculation required to identify the ODG portion corresponding to an event. The "trigger events (b)" in Figure 9 correspond to this portion of the ODG.

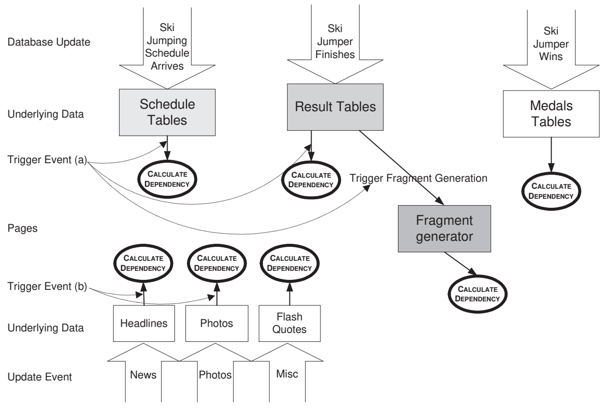

Figure 9: The calculated portion of the ODG.

Finally, it may not be possible to calculate some components of the object dependence graph directly from the initial event. These components tend to be dynamic and change over the course of an event. An example would be athlete and country pages. A given athlete page is only dependent on changes in the medals tables if the athlete wins a medal; a country page is not dependent on medals or recent results updates until its athletes actually perform. These dependencies are discovered and calculated over time as results arrive. Recent news and results are embedded into appropriate pages as HTML fragments which were generated from triggers as described above. Since fragment generation might modify the dependency relationships residing in the cache (e.g. Germany's page is dependent on updates to the medals table), the pages composed of the fragments must also be generated when the request completes. The fragments constitute both underlying data and objects. We therefore view building fragments as one step in calculating the ODG for the initial event. We represent this in Figure 9 as the "Trigger Fragment Generation" action that occurs in response to a trigger event. Combining Figures 8 and 9, we get the completed ODG of Figure 10.

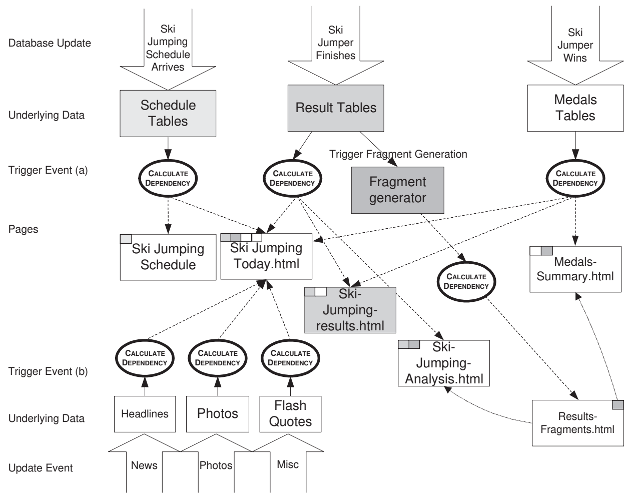

Figure 10: The complete ODG.

After expansion of the ODG and generation of HTML, pages and fragments are scanned for error messages and are discarded and retried until the pages are correct, or until it is determined (by retry count or manual intervention) that the page cannot be built correctly. The final action of the trigger monitor is to insure that generated pages are successfully delivered to the servers. Commit and retry algorithms are used to insure that temporary malfunction of a server does not result in data inconsistency.

3.4 Maintaining Consistency Among Pages

A Web site is composed of pages which may result from several discrete but related data sources. Data changes may affect multiple pages, and insuring that the pages are mutually consistent can be difficult. A number of consistency problems arose in the Olympic Games:

- Any given fragment or page is typically composed from multiple database tables, and any given event typically causes multiple tables to be updated. How does one determine which database updates should trigger generation, and how does one eliminate multiple redundant page generations?
- Most pages consist of multiple fragments, and often a single database update results in many fragment updates. How does one insure that all fragments have been generated before the page is generated?
- News stories and headlines are published on separate pages. How can one insure, in the presence of propagation and update delays, that headlines are not physically made available until all related news stories are also available?

The first problem is solved with the *trigger table*. The *trigger table* solves several interesting problems in addition to consistency and is discussed in detail in Section 3.5.

The second problem is solved by permitting a single trigger event to be processed in any number of “passes”. Each pass through the processing loop is equivalent to redelivery of the same trigger with the correct pass number. Mechanisms are implemented to easily tailor the page generation logic for each pass through the processing loop. Thus, one might generate all fragments associated with a trigger in the first pass, and all pages composed of those fragments in the second pass.

The third problem is solved by creating a *dependent group* of triggers. A *dependent group* consists of exactly one *independent* trigger followed by zero or more *dependent* triggers. Dependent groups are processed one group at a time in strict FIFO order. We refer to *independent* triggers as type *S* triggers because independent triggers are processed sequentially. *Dependent* triggers are referred to as type *P* triggers because all dependent triggers within a group may be processed in parallel.

Within a dependent group, the independent type *S* trigger must be fully processed before any of the dependent type *P* triggers are processed. Since *dependent groups* are processed in FIFO order, this implies that at any given time, at most one type S trigger is being processed. On completion of a *dependent group*, the data source is optionally notified of completion of the group in order to aid the data source in sequencing triggers appropriately.

For every news story that arrives, many sets of headlines require updating because multiple hypertext links to the story exist. No headline can be updated until all relevant stories are also published. In order to provide stories and headlines in the proper order, news is published in dependent groups consisting of a news story (the independent type *S* trigger) followed by all relevant headlines (the dependent type *P* triggers). If a set of headlines are dependent on multiple news stories, each news story corresponds to an *S* trigger of a different dependent group; the headlines correspond to *P* triggers of the last dependent group.

The technique is applicable to other types of problems, for example, insuring that product descriptions do not get published before the related product details have been published at an e-commerce site.

3.5 The Trigger Table

In DB2/6000 which was used to store underlying data at the Olympic Games Web site, a programmer may write SQL statements which are automatically executed when changes occur in the database. This SQL code is known as a *trigger*. Triggers may be associated with a wide variety of changes such as row insertion, update, or deletion. The *trigger table* used in the Olympic Games was an ordinary DB2 table containing triggers which were executed whenever a row was inserted into the trigger table.

We used the trigger table for several purposes: synchronizing data updates with generated pages, guaranteeing delivery of triggers, and permitting Lotus Notes to interact with the trigger monitor without any direct knowledge of the trigger monitor API or message format.

Note that the trigger monitor itself does not require a database or database triggers, and in fact has no knowledge of databases at all. The trigger monitor mechanism is encapsulated entirely in the trigger monitor API and may be invoked by any entity within the system (e.g. as a result of firing a database trigger).

3.5.1 Synchronizing Updates

If the database is normalized, any single external event (e.g. somebody winning a sporting event, a price change in a commerce site) typically results in the update of many different tables. Although one could trigger page generation on update of one or more tables, this can be difficult to do correctly and efficiently. How does one determine when all tables are updated? How does one prevent redundant regenerations of the same page? Database transactions associated with an event may change over time. If new table updates are added or existing table updates are removed from a transaction, how does one insure that the correct triggers continue to be delivered?

The trigger table solves all of these problems. Each row of the trigger table corresponds to a database transaction which may cause multiple updates to database tables. We assume that the data loader knows when a database transaction is complete and thus when to commit each transaction. A trivial modification to the loader enabled it to update the trigger table after each transaction was committed. Data in the trigger table encoded a summary of the related transaction such as the event type, date, status, and phase. Figure 11 depicts an abstract view of this process. Figure 6 shows how the trigger table fits in the overall system.

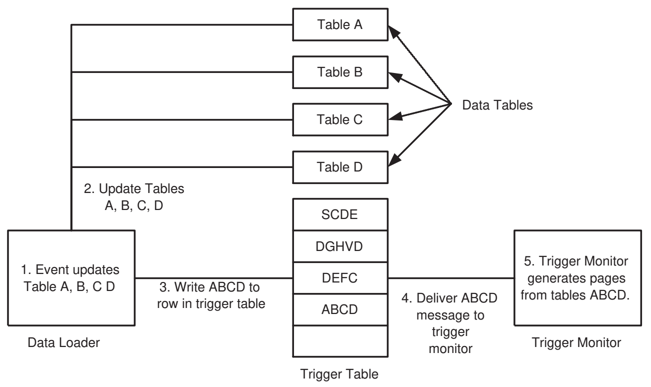
Table A
Table B
Table C
Table D
Data Tables
2. Update Tables A, B, C, D
SCDE
DGHVD
DEFC
ABCD
1. Event updates Table A, B, C D
3. Write ABCD to row in trigger table
4. Deliver ABCD message to trigger monitor
5. Trigger Monitor generates pages from tables ABCD.
Data Loader
Trigger Table
Trigger Monitor

Figure 11: The trigger table.

Each database trigger is thus attached to a trigger table row and not the tables involved in the transaction. A single trigger can update pages after multiple table updates corresponding to a single committed transaction are performed, avoiding redundant page regenerations. One may change the transaction without affecting triggers, since the triggers are defined at the transaction level, not on the individual tables.

3.5.2 Guaranteed Delivery

It is important that a trigger, once issued by the application, is

1. accepted quickly, and
2. once accepted, is guaranteed to be carried through to completion.

Rather than attempting to encapsulate complex and potentially expensive protocols into the trigger monitor API, the system uses DB2/6000 triggers to guarantee delivery. One field in each row of the trigger table is used to record when the trigger monitor has accepted each trigger. Committing the row in the trigger table provides a permanent record of the event. The database can easily be queried to determine which triggers have been delivered and which have not. Once in the database, it is a trivial matter to deliver or redeliver any particular trigger with no knowledge or assistance from the application.

3.5.3 Application Independence

One interesting use of DB2 triggers is that ODBC calls can be used to insert rows into the trigger table and thus notify the trigger monitor of data changes without knowing about the trigger monitor API itself. This allows ODBC-enabled applications to use the triggering mechanisms without modifications to the source code.

We used this technique in the Olympic Games as the basis for the interface between Lotus Notes and the trigger monitor. News stories were edited by sports writers using Lotus Notes. When a news story was published, an ODBC request was issued by Lotus Notes to DB2 to insert the Lotus URL for the story into the trigger table. A DB2 trigger transmitted the URL to the trigger monitor which then fetched the story and placed it in the Web server's document directory. These documents were now served to the Internet as flat files. We thus combined the communication, editing, and publishing capabilities of Lotus Notes with the high throughput achievable when serving static files.

4 Performance of the 1998 Olympic Games Web Site

As a result of DUP and prefetching, we were able to achieve cache hit rates close to 100% throughout the entire Olympic Games. These high hit rates allowed the system to serve pages quickly even during times of peak demand.

The site contained approximately 87,000 unique pages of which approximately 21,000 were dynamically generated. Dynamic pages reflected current events within a maximum of sixty seconds after the events occurred. Up to 58,000 pages were generated or regenerated per day during peak activity, with an average of 20,000 pages generated per day over the course of the Olympic Games. All dynamic pages could be cached in memory without overflow. Therefore, the system never had to apply a cache replacement algorithm. In general, the memory requirements for caching data belonging to a single Web site are significantly less than those required for proxy caches which store data from several Web sites [5, 2, 8]. The maximum memory required for a single copy of all cached objects was around 175 Mbytes. A detailed description of the architecture of the Olympic Games Web site is contained in [3].

A total of 634.7 million requests were serviced during the Olympic Games. On the peak day (Day 7, Feb 13), the site served 56.8 million requests over a 24-hour period. By contrast, the 1996 Olympic Games Web site peaked at 17 million hits a day, fewer than any day for the 1998 Olympic Games. The maximum number of hits per minute was 110,414; this occurred around the time of the Women's Figure Skating Free Skating on February 20 (Day 14). Even during peak request periods such as this one, the Web site was able to serve data at all times without ever coming close to sustaining maximum load.

The total number of hits to the site as well as the maximum number of hits during a single minute were both determined by independent organizations which audited the Web logs. On July 14, 1998, the Guinness Book of World Records recognized the 1998 Olympic Games Web site for setting two world records:

- *The Most Popular Internet Event Ever Recorded* based on the officially audited figure of 634.7 million requests over the 16 days of the Olympic Games.
- *The Most Hits On An Internet Site In One Minute* based on the officially audited figure of 110,414 hits received in a single minute around the time of the Women's Figure Skating Free Skating.

Figure 12 shows the number of hits received by the Web site each day. Table 1 compares the access times and transmission rates for the Olympic Games Web site to those of home pages of other major US Web sites measured on Day 14 in the US using 28.8 Kbps modems. These numbers indicate that the Nagano site was one of the most responsive sites on the Internet. Response times for the Olympic Games Web site measured in other countries were also fast [3].

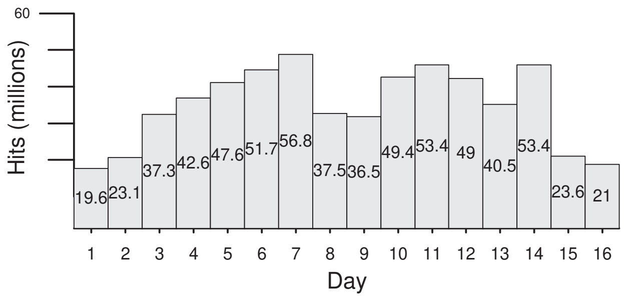

Figure 12: Hits by day in millions.

Even during peak periods, the system was never close to being stressed. Virtually all of the delays in response times for the Olympic Games Web site in Table 1 were caused not by the Web site but by the client and the client connection to the network. For clients communicating with the Internet via fast links, response times were nearly instantaneous. Additional performance statistics for the Web site are contained in [3].

5 Conclusion

We have presented DUP which is a new approach for consistently caching dynamic Web data in order to improve performance. DUP maintains data dependence information between cached objects and the underlying data which affect their values in a graph. When the system becomes aware of a change to underlying data, graph traversal algorithms are applied to determine which cached objects are affected by the change. Cached objects which are found to be highly obsolete are then either invalidated or updated.

We described how DUP was implemented at the official Web site for the 1998 Olympic Winter Games. By using DUP, we were able to achieve cache hit rates close to 100% compared with 80% for an earlier version of our system which did not employ DUP. As a result of the high cache hit rates, the Olympic Games Web site was able to serve data quickly even during peak request periods.

Acknowledgements

Many people made valuable contributions to the 1998 Olympic Games Web site including Kim Benson, John Chiavelli, Glen Druce, Cameron Ferstat, Kip Hansen, Michael Maturo, Kent Rankin, Paul Reed, Jerry Spivak, Elin Stilwell, Brian Taylor, and John Thompson.

Web Site	Olympic Games	Compuserve	AOL	MSN	NETCOM	AT&T
Mean Response Time (seconds)	18.26	19.14	23.91	20.17	19.72	19.71
Transmit Rate (Kbps)	23.31	21.86	19.05	18.60	21.01	20.84

Table 1: Response times and transmission rates measured in the US on Day 14 using 28.8 Kbps modems. These numbers were measured by IBM and were not verified by an independent organization.

References

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

[2] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[3] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proceedings of ACM/IEEE SC98*, November 1998.

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

[5] B. M. Duska, D. Marwood, and M. J. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[6] 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.

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

[8] S. Manley and M. Seltzer. Web Facts and Fantasy. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.