A Publishing System for Efficiently Creating Dynamic Web Content

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

Cameron Ferstat, Paul Reed
IBM Global Services
17 Skyline Drive
Hawthorne, NY 10532

Abstract— **This paper presents a publishing system for efficiently creating dynamic Web content. Complex Web pages are constructed from simpler fragments. Fragments may recursively embed other fragments. Relationships between Web pages and fragments are represented by object dependence graphs. We present algorithms for efficiently detecting and updating Web pages affected after one or more fragments change. We also present algorithms for publishing sets of Web pages consistently; different algorithms are used depending upon the consistency requirements.**

Our publishing system provides an easy method for Web site designers to specify and modify inclusion relationships among Web pages and fragments. Users can update content on multiple Web pages by modifying a template. The system then automatically updates all Web pages affected by the change. Our system accommodates both content that must be proofread before publication and is typically from humans as well as content that has to be published immediately and is typically from automated feeds.

I. INTRODUCTION

Many Web sites need to provide dynamic content. Examples include sport sites [2], stock market sites, and virtual stores or auction sites where information on available products is constantly changing.

There are several problems with providing dynamic data to clients efficiently and consistently. A key problem with dynamic data is that it can be expensive to create; a typical dynamic page may require several orders of magnitude more CPU time to serve than a typical static page of comparable size. The overhead for dynamic data is a major problem for Web sites which receive substantial request volumes. Significant hardware may be needed for such Web sites.

A key requirement for many Web sites providing dynamic data is to completely and consistently update pages which have changed. In other words, if a change to underlying data affects multiple pages, all such pages should be correctly updated. In addition, a bundle of several changed pages may have to be made visible to clients at the same time. For example, publishing pages in bundles instead of individually may prevent situations where a client views a first page, clicks on a hypertext link to view a second page, and sees information on the second page which is older and not consistent with the information on the first page.

Depending upon the way in which dynamic data are being served, achieving complete and consistent updates can be difficult or inefficient. Many Web sites cache dynamic data in memory or a file system in order to reduce the overhead of recalculating Web pages every time they are requested [7]. In these systems, it is often difficult to identify which cached pages are affected by a change to underlying data which modifies several dynamic Web pages. In making sure that all obsolete data are invalidated, deleting some current data from cache may be unavoidable. Consequently, cache miss rates after an update may be high, adversely affecting performance. In addition, multiple cache invalidations from a single update must be made consistently.

This paper presents a system for efficiently and consistently publishing dynamic Web content. In order to reduce the overhead of generating dynamic pages from scratch, our system composes dynamic pages from simpler entities known as *fragments*. Fragments typically represent parts of Web pages which change together; when a change to underlying data occurs which affects several Web pages, the fragments affected by the change can easily be identified. It is possible for a fragment to recursively embed another fragment.

Our system provides a user-friendly method for managing complex Web pages composed of fragments. Users specify how Web pages are composed from fragments by creating templates in a markup language. Templates are parsed to determine inclusion relationships among fragments and Web pages. These inclusion relationships are represented by a graph known as an *object dependence* *graph (ODG)*. Graph traversal algorithms are applied to ODG's in order to determine how changes should be propagated throughout the Web site after one or more fragments change.

Our system allows multiple independent authors to provide content as well as multiple independent proofreaders to approve some pages for publication and reject others. Publication may proceed in multiple stages in which a set of pages must be approved in one stage before it is passed to the next stage. Our system can also include a link checker which verifies that a Web page has no broken hypertext links at the time the page is published.

A key feature of our system is that it is scalable to handle high request rates. We are planning to deploy versions of our system at popular sports Web sites such as the 2000 Olympic Games Web site.

The remainder of the paper is organized as follows. Section II describes the architecture of our system in detail. Section III describes the performance of our system. Section IV discusses related work. Finally, Section V summarizes our main results and conclusions.

II. System Architecture

A. Constructing Web Pages from Fragments

A.1 Overview

A key feature of our system is that it composes complex Web pages from simpler fragments (Figure 8). A page is a complete entity which may be served to a client. We say that a fragment or page is *atomic* if it doesn't include any other fragments and *complex* if it includes other fragments. An *object* is either a page or a fragment.

Our approach is efficient because the overhead for composing an object from simpler fragments is usually minor. By contrast, the overhead for constructing the object from scratch as an atomic fragment is generally much higher. Using the fragment approach, it is possible to achieve significant performance improvements without caching dynamic pages and dealing with the difficulties of keeping caches consistent. For optimal performance, our system has the ability to cache dynamic pages. Caching capabilities are integrated with fragment management.

The fragment-based approach for generating Web pages makes it easier to design Web sites in addition to improving performance. It is easy to design a set of Web pages with a common look and feel. It is also easy to embed common information into several Web pages. Sets of Web pages containing similar information can be managed together. For example, it is easy to update common information represented by a single fragment but embedded within multiple pages; in order to update the common information everywhere, only the fragment needs to be changed.

By contrast, if the Web pages are stored statically in a file system, identifying and updating all pages affected by a change can be difficult. Once all changed pages have been identified, care must be taken to update all changed pages in order to preserve consistency.

Dynamic Web pages which embed fragments are implicitly updated any time an embedded fragment changes, so consistency is automatically achieved. Consistency becomes an issue with the fragment-based approach when the pages are being published to a cache or file system. Our system provides several different methods for consistently publishing Web pages in these situations; each method provides a different level of consistency.

A.2 Object Dependence Graphs

When pages are constructed from fragments, it is important to construct a fragment f_1 before any object containing f_1 is constructed. In order to construct objects in an efficient order, our system represents relationships between fragments and Web pages by graphs known as *object dependence graphs* (ODG's) (Figures 1 and 2).

Object dependence graphs may have several different edge types. An *inclusion edge* indicates that an object embeds a fragment. A *link edge* indicates that an object contains a hypertext link to another object.

In the ODG in Figure 2, all but one of the edges are inclusion edges. For example, the edge from f_4 to P_1 indicates that P_1 contains f_4; thus, when f_4 changes, f_4 should be updated before P_1 is updated. The graph resulting from only inclusion edges is a directed acyclic graph.

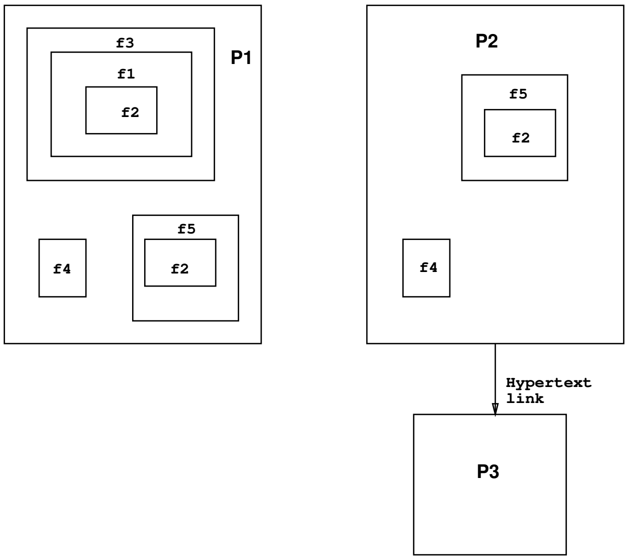

Fig. 1. A set of Web pages containing fragments.

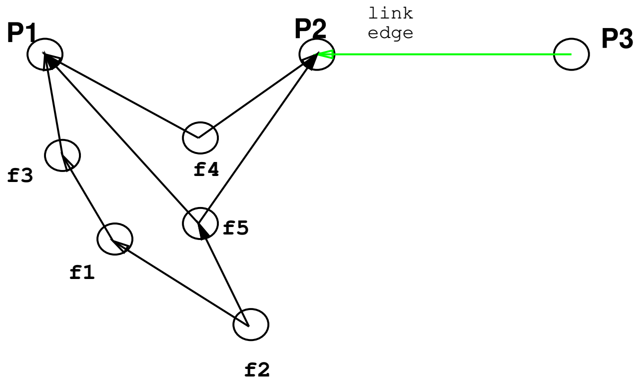

Fig. 2. The object dependence graph (ODG) corresponding to Figure 1.

The edge from P_3 to P_2 is a link edge which indicates that P_2 contains a hypertext link to P_3. A key reason for maintaining link edges is to prevent dangling or inconsistent hypertext links. In this example, the link edge from P_3 to P_2 indicates that publishing P_2 before P_3 will result in a broken hypertext link. Similarly, when both P_2 and P_3 change, publishing a current version of P_2 before publishing a current version of P_3 could present inconsistent information to clients who view an updated version of P_2, click on the hypertext link to an outdated version of P_3, and then see information which is obsolete relative to the referring page. Link edges can form cycles within an ODG. This would occur, for example, if two pages both contained hypertext links to each other.

There are two methods for creating and modifying ODG's. Using one approach, users specify how Web pages are composed from fragments by creating templates in a markup language. Templates are parsed to determine inclusion relationships among fragments and Web pages. Using the second approach, a program may directly manipulate edges and vertices of an ODG by using an API.

Our system allows an arbitrary number of edge types to exist in ODG's. So far, we have only found practical use for inclusion and link edges. We suspect that there may be other types of important relationships which can be represented by other edge types.

When our system becomes aware of changes to a set S of one or more objects, it does a depth-first graph traversal using topological sort [4] to determine all vertices reachable from S by following inclusion edges. The topological sort orders vertices such that whenever there is an edge from a vertex v to another vertex u, v appears before u in the topological sort. For example, a valid topological sort of the graph in Figure 2 after P_3, f_4, and f_2 change would be P_3, f_4, f_2, f_5, P_2, f_1, f_3, and P_1. This topological sort ignores link edges.

Objects are updated in an order consistent with the topological sort. Our system updates objects in parallel when possible. In the previous example, P_3, f_4, and f_2 can be updated in parallel. After f_2 is updated, f_1 and f_5 may be updated in parallel. A number of other objects may be constructed in parallel in a manner consistent with the inclusion edges of the ODG.

After a set of pages, U, has been updated (or generated for the first time), the pages in U are published so that they can be viewed by clients. In some cases, the pages are published to file systems. In other cases, they are published to caches. Pages may be published either locally on the system generating them or to a remote system. It is often a requirement for a set of multiple pages to be published consistently. Consistency can be guaranteed by publishing all changed (or newly generated) pages in a single atomic action. One potential drawback to this method of publication is that the publication process may be relatively long. For example, pages may have to be proofread before publication. If everything is published together in a single atomic action, there can be considerable delay before any information is made available.

Therefore, incremental publication, wherein information is published in stages instead of together, is often desirable. The disadvantage to incremental publication is that consistency guarantees are not as strong. Our system provides three different methods for incremental publication, each providing different levels of consistency.

The first incremental publishing method guarantees that a freshly published page will not contain a hypertext link to either an obsolete or unpublished page. This consistency guarantee applies to pages reached by following several hypertext links. More specifically, if P_1 and P_2 are two pages in U, if a client views an updated version of P_1 and follows one or more hypertext links to view P_2, then the client is guaranteed to see a version of P_2 which is not obsolete with respect to the version of P_1 which the client viewed (a version of P_2 is obsolete with respect to a version of P_1 if the version of P_2 was outdated at the time the version of P_1 became current, regardless of whether P_1 or P_2 have any fragments in common).

For example, consider the Web pages in Figure 3. A client can access P_3 by starting at P_1, following a hypertext link to P_2 and then following a second hypertext to P_3. Suppose that both P_1 and P_3 change. The first incremental publishing method guarantees that the new version of P_1 will not be published before the new version of P_3, regardless of whether P_2 has changed.

This incremental publishing method is implemented by first determining the set R of all pages which can be reached by following hypertext links from a page in U. R includes all pages of U; it may also include previously published pages which haven't changed. R is de-

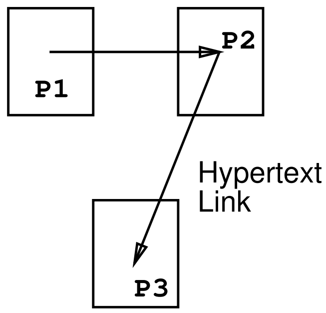

Fig. 3. A set of Web pages connected by hypertext links.

termined by traversing link edges in reverse order starting from pages in U.

Let K be the subgraph of the ODG consisting of all nodes in R and link edges in the ODG connecting nodes in R. K is topologically sorted, and its strongly connected components are determined. A strongly connected component of a directed graph is a maximal subset of vertices S such that every vertex in S has a directed path to every other vertex in S. A good algorithm for finding strongly connected components in directed graphs is contained in [4].

Vertices in U are then examined in an order consistent with the topological sort of K. Each time a page in U is examined for which the updated version hasn't been published yet, the page is published together with all other pages in U belonging to the same strongly connected component. Each set of pages which are published together in an atomic action is known as a *bundle.*

The second incremental publishing method guarantees that any two pages in U which both contain a common changed fragment are published in the same bundle. For example, consider the Web pages in Figure 4. Suppose that both f_1 and f_2 change. Since P_1 and P_3 both embed f_1, their updated versions must be published together. Since P_2 and P_3 both embed f_2, their updated versions must be published together. Thus, updated versions of all three Web pages must be published together. Note that updated versions of P_1 and P_2 must be published together, even though the two pages don't embed a common fragment.

In order to implement this approach, the set of all changed fragments contained within each changed object d_1 is determined. We call this set the *changed fragment set* for d_1 and denote it by $C(d_1)$. All changed objects are constructed in topological sorting order. When a changed object d_1 is constructed, $C(d_1)$ is calculated as the union of f_2 and $C(f_2)$ for each fragment f_2 such that a dependence edge (f_2, d_1) exists in the ODG.

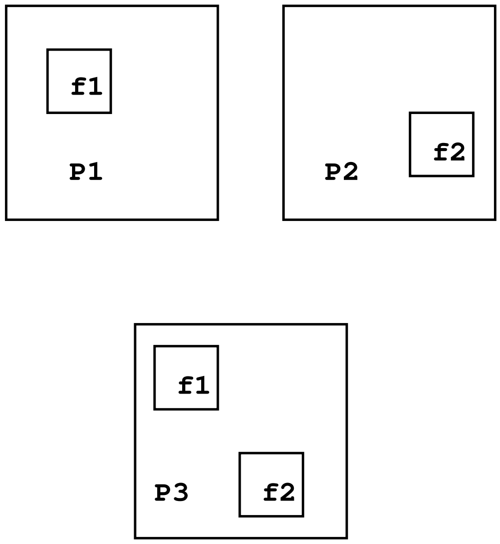

Fig. 4. A set of Web pages containing common fragments.

After all changed fragment sets have been determined, an undirected graph D is constructed in which the vertices of D are pages in U. An edge exists between two pages P_1 and P_2 in U if $C(P_1)$ and and $C(P_2)$ have at least one fragment in common. D is examined to determine its connected components (two vertices are part of the same connected component if and only if there is a path between the vertices in the graph). All pages belonging to the same connected component are published in the same bundle.

The third incremental publishing method satisfies the consistency guarantees of both the first and second method. In other words,

1. A freshly published page will not contain a hypertext link to either an obsolete or unpublished page. More specifically, if P_1 and P_2 are two pages in U, if a client views an updated version of P_1 and follows one or more hypertext links to view P_2, then the client is guaranteed to see a version of P_2 which is not obsolete with respect to the version of P_1 which the client viewed.
2. Any two changed pages which both contain a common changed fragment are published together.

This method generally results in publishing fewer bundles but of larger sizes than the first two approaches.

For example, consider the Web pages in Figure 5. Suppose that both P_1 and f_1 change. Updated versions of P_2 and P_3 must be published together because they both embed f_1. Since P_1 contains a hypertext link to P_3, the updated version of P_1 cannot be published before the bundle containing updated versions of P_2 and P_3.

If, instead, the first incremental publishing method were used to publish the Web pages in Figure 5, the updated version of P_1 could not be published before the updated version of P_3. However, the updated version of P_2 would

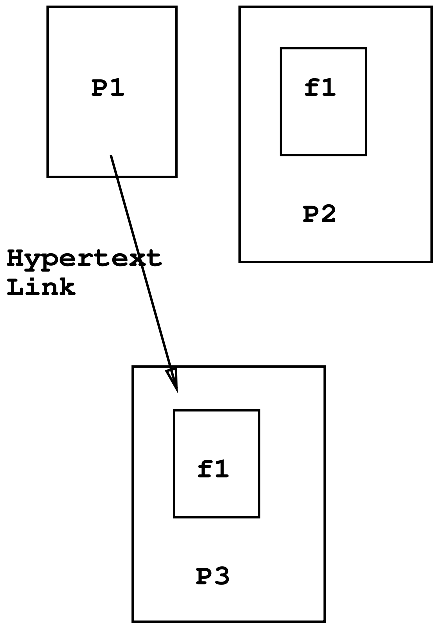

Fig. 5. Another set of related Web pages.

not have to be published in the same bundle as the updated version of P_3. If the second incremental publishing method were used, updated versions of both P_2 and P_3 would have to be published together in the same bundle. However, publication of the updated version of P_1 would be allowed to precede publication of the bundle containing updated versions of P_2 and P_3.

The third incremental publishing method is implemented by constructing K as in the first incremental publishing method and changed fragment sets as in the second incremental publishing method. Additional edges are then added to K between pages in U. For all pages P_1 and P_2 in U such that $C(P_1)$ and $C(P_2)$ have a fragment in common, directed edges from both P_1 to P_2 and P_2 to P_1 are then added. The same procedure is then applied to K to publish pages in bundles as in the first method.

Incremental publishing methods can be designed for other consistency requirements as well. For example, consider Figure 3. Suppose that both P_1 and P_3 change. It may be desirable to publish updated versions of P_1 and P_3 in the same bundle. This would avoid the following situation which could occur using the first incremental publishing method.

A client views an old version of P_1. After following hypertext links, the client arrives at a new version of P_3. The browser's cache is then used to go to the old version of P_1. The client reloads P_1 in order to obtain a version consistent with P_3 but still sees the old version because the new version of P_1 has not yet been published.

It is straightforward to implement an incremental publishing method which would publish P_1 and P_3 in the same bundle using techniques similar to the ones just described.

B. The Publishing System

B.1 Combined Content Pages

Many of the news, sports, and entertainment Web sites that IBM produces contain information that is fed from multiple sources. Some of the information, such as the latest scores from a sporting event, is generated automatically by a computer. Other information, such as news stories, is generated by humans. Both types of information are subject to change. A page containing both human and computer-generated information is known as a *combined content page.*

A key problem with serving combined content pages is the different rates at which sources produce content. Computer-generated content tends to be produced at a relatively high rate, often as fast as the most sophisticated timing technology permits. Human-generated content is produced at a much lower rate. Thus, it is difficult for humans to keep pace with automated feeds. By the time an editor has finished with a page, the actual results on the page may have changed. If the editor takes time to update the page, the results may have changed yet again.

A requirement for many of the Web sites we have helped design is that computer-generated content should not be delayed by humans. Computer-generated results such as the latest results from a sporting event are often extremely important and should be published as soon as possible. If computer-generated results are combined with human-edited content using conventional Web publishing systems, publication of the computer-generated results can be delayed significantly. What is needed is a scheme to combine data feeds of differing speeds so that information arriving at high rates is not unnecessarily delayed.

In order to provide combined content pages, our system divides fragments into two categories. *Immediate fragments* are fragments which contain vital information which should be published quickly with minimal proofreading. For the sports Web sites that our system is being used for, the latest results in a sporting event would be published as an immediate fragment. *Quality controlled fragments* are fragments which don't have to be published as quickly as immediate fragments but have content which must be examined in order to determine whether the fragments are suitable to be published. Background stories on athletes are typically published as quality controlled fragments at the sports sites which use our system. Combined content Web pages consist of a mixture of immediate and quality controlled fragments.

When one or more immediate fragments change, the Web pages affected by the changes are updated and published without proofreading. If both immediate and quality controlled fragments change, the system first performs updates resulting from the immediate fragments and publishes the updated Web pages immediately. It subsequently performs updates resulting from quality controlled fragments and only publishes these updated Web pages after they have been proofread. Multiple versions of a combined content page may be published using this approach. The first version would be the page before any updates. The second version might contain updates to all immediate fragments but not to any quality controlled fragments. The third version might contain updates to all fragments.

It is possible for an update to an immediate fragment f_1 to be published before an update to a quality controlled fragment f_2 even though f_2 changed before f_1. This might occur if the changes to f_2 are delayed in publication due to proofreading.

B.2 System Description

Web pages produced by our system typically consist of multiple fragments. Each fragment may originate from a different *source* and may be produced at a different rate than other fragments. Fragments may be nested, permitting the construction of complex and sophisticated pages. Completed pages are written to *sinks,* which may be file systems, Web server accelerators [9], or even other HTTP servers.

The Trigger Monitor is the software which takes objects from one or more sources, constructs pages, and writes the constructed pages to one or more sinks (Figure 6). Relationships between fragments are maintained in a persistent ODG which preserves state information in the event of a system crash. Our new Trigger Monitor has significantly enhanced functionality compared with the Trigger Monitor used for the 1998 Olympic Games Web site [2], [3].

Whenever the Trigger Monitor is notified of a modification, addition, or deletion of one or more objects, it fetches new copies of the changed objects from one or more sources. The ODG is updated by parsing changed objects. The graph traversal algorithms described in Section II-A.2 are then applied to determine all Web pages which need to be updated and an efficient order for updating them. Finally, bundles of published pages are written to the sinks.

Since the Trigger Monitor is aware of all fragments and pages, synchronization is possible to prevent corruption of the pages. The ODG is used as the synchronization object to keep the fragment space consistent. Many “trigger handlers”, each with their own sources and sinks, may

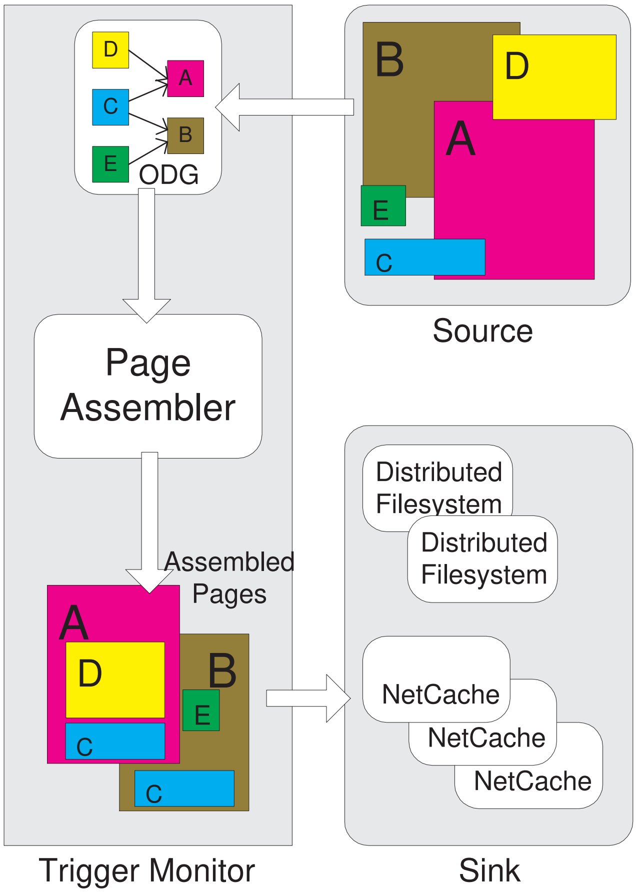

Fig. 6. Schematic of the Publish Process.

be configured to use a common ODG. This design permits, for example, a slow-moving, carefully edited human-generated set of pages and fragments to be integrated with a high-speed, automated, database-driven content source. Because the ODG is aware of the entire fragment space and the interrelationship of the objects within that space, synchronization points can be chosen to ensure that multiple, differently-sourced, differently-paced content streams remain consistent.

Multiple Trigger Monitor instances may be chained, the sinks of earlier instances becoming the sources for later ones. This allows publication to take place in multiple stages. For example, the publishing system for the 2000 Summer Olympic Games Web site will consist of the following stages (Figure 7):

Development is the first step in the process. Fragments which appear on many Web pages (such as generic headers and footers) as well as overall site design occur here. The output of development may be structurally complete but lacking in content.

Staging takes as its input, or *source,* the output, or *sink,* of Development. Editors polish pages and combine content from various sources. Finished pages are the result.

Quality Assurance takes as its source the *sink* of Staging. Pages are examined here for correctness (grammatical, factual, and political) and appropriateness.
Production is where pages are served from. Its source is the *sink* of QA, and its *sinks* are the serving directories and caches.
Automated Results are produced when a database trigger is generated as the result of an update. The trigger causes programs to be executed that extract current results and compose relevant updated pages and fragments. Unlike the previous stages, no human intervention occurs in this stage.

Note how one stage can use the sink of another stage as its source. The automated feed updates each source at the same time, but independently of the human-driven stages. This achieves the dual goals of keeping the entire site consistent while publishing content immediately from automated feeds.

A similar organization was used for the 1998 Winter Olympic Games Web site. The primary difference was that the process of moving pages from one stage to the next was purely manual. In other words, authors had to keep track of all the pages that were affected by their changes and move them down to Staging, editors had to move their material to Q/A, and so on. This required the authors to know something about the editing process and the editors to know about Q/A. Learning the process was difficult enough; changing it was even worse.

Our new system eliminates most of the procedural difficulties which were experienced at the 1998 Olympic Games Web site. Stages can be added and deleted easily. Data sources can be added and deleted with little or no disruption to the flow. The new system adapts much more easily to changing conditions and requires people working on specific stages of the system to know less about what is required for other stages.

C. Example

To demonstrate how a site might be built from fragments, we present a real example from the official Web site for the 1999 French Open Tennis Tournament. A site architect views the player page for Steffi Graf (shown in Figure 8) as consisting of a standard header, sidebar, and footer, with biographical information and recent results thrown in. The site architect composes HTML similar to the following, establishing a general layout for the site:

```
<html>
<!-- %include(header.frg) -->
  <table>
    <tr>
      <td><!-- %include(sidebr.frg) --></td>

      <td><table>
        <tr><!-- %fragment(graf_bio.frg) --></tr>
        <tr><!-- %fragment(graf_score.frg) --></tr>
      </td></table>

    </tr>
  </table>
<!-- %include(footer.frg) -->
</html>
```

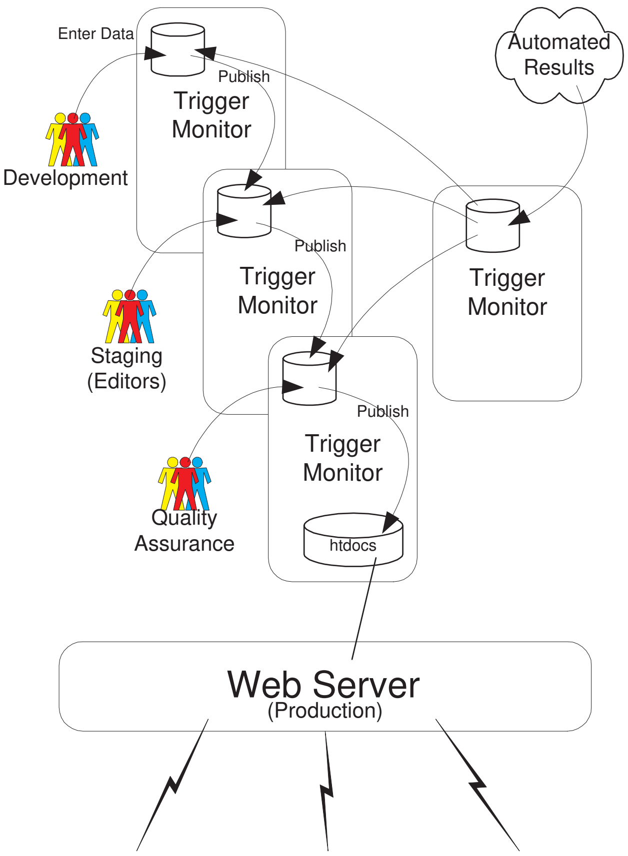

Fig. 7. Schematic of the Publish Process.

where "footer.frg" consists of

```
<!-- %fragment(factoid.frg) -->
<!-- %fragment(copyr.frg) -->
```

Prior to the beginning of play, the contents of "graf_score.frg" will be empty, since no matches have commenced. This means the part of the page outlined by the dashed box in Figure 8 will, at first, be empty. The first publication of this fragment will result in the ODG seen to the right of Steffi Graf's player page in Figure 8. Again, the objects and edges within the dashed box will not yet be within the ODG, since no match play has yet occurred.

Using fragments in this way permits many architects, editors, and even automated systems to modify the page simultaneously. Our system ensures that all changes are properly included in the final page that is seen by the user. An architect updating the structure of the page does not need to know anything about copyrights, trademarks, the

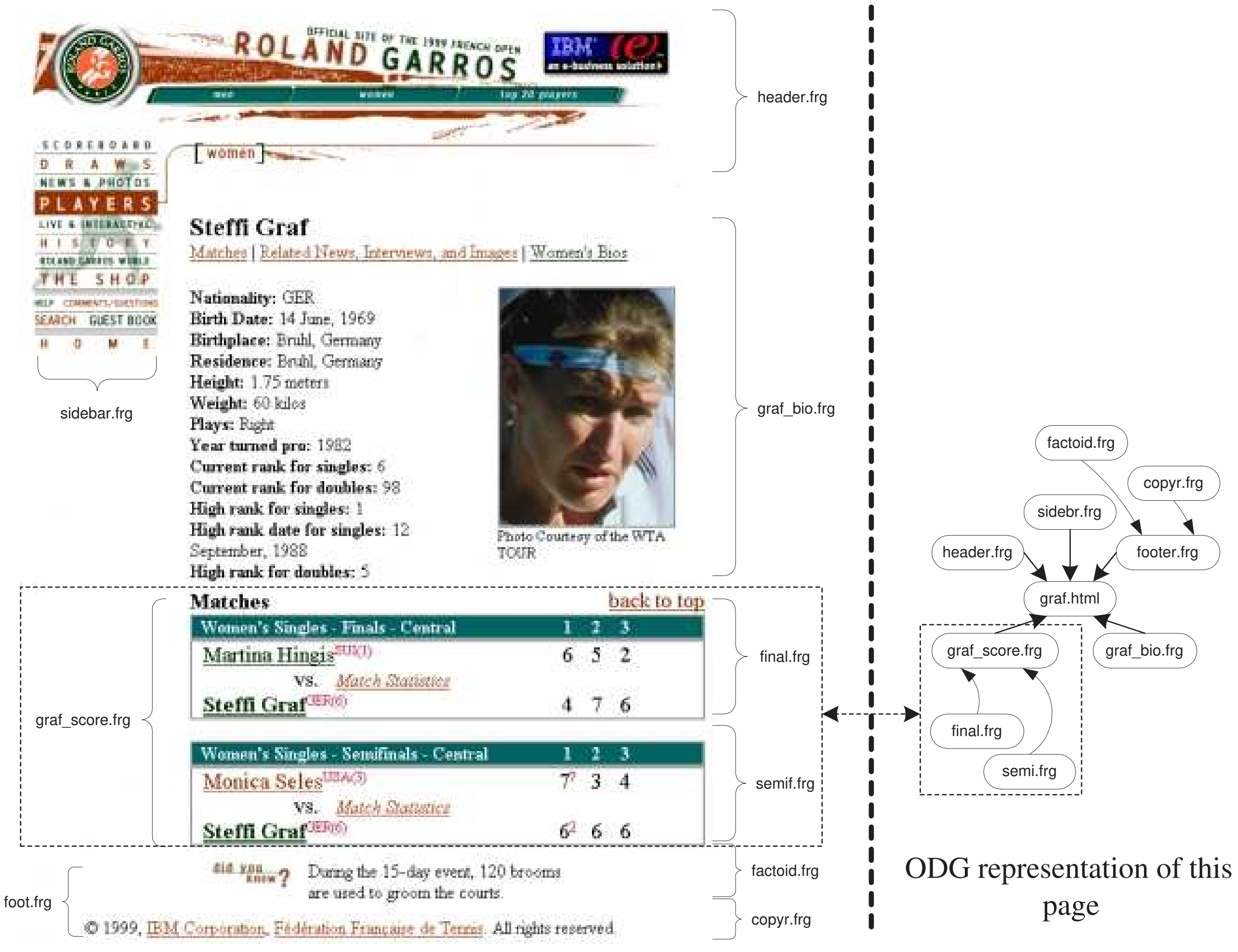

Fig. 8. Sample screen shot from the official Web site for the French Open Tennis Tournament.

size of the sponsor's logos, the look-and-feel of the site, or any of the data that will be included on the page. Similarly, an editor wishing to change the look-and-feel of a site does not need to understand the structure of any particular page.

Major site changes, like changing the look-and-feel of a site, are as simple as changing a single page. For example, changing the sidebar to reflect the end of a long event is as simple as updating "sidebr.frg". To change the look-and-feel of a site, an editor only needs to change "header.frg" and "footer.frg". For both these kinds of changes, the system will use the ODG from Figure 8 to determine that Steffi Graf's page must be rebuilt (along with many others). Once all pages have been rebuilt, they will be republished. The user will see the changes on every page, although the vast majority of underlying fragments will not have changed.

More static information, like player biographies, can be kept up-to-date in one place but used on many pages. For example, "graf_bio.frg" is used on our example page, but may also be used in many other places. To include a new photo or update the information included in the biography, the editors need only concern themselves with updating "graf_bio.frg". The system ensures that all pages which include "graf_bio.frg" will automatically be rebuilt.

Since scoring information will change frequently once a tennis match is in progress, updating that aspect of a page can be handled by an automated process. As a match begins, "graf_score.frg" is updated to include the match in progress. This means that once the final has begun, the "graf_score.frg" page will consist of HTML similar to

```
<!-- %fragment(final.frg) -->
<!-- %fragment(semi.frg) -->
```

When the updated "graf_score.frg" is published, the system will detect that it now includes "final.frg" and "semi.frg" and will update the ODG as shown in the dashed box within Figure 8. Now, as the final match progresses, only "final.frg" needs to be updated and published through our system. As part of the publication process, the system will detect that "final.frg" is included in "graf_score.frg", causing "graf_score.frg" to be rebuilt using the updated score. Likewise, the system will detect that Steffi Graf's page must be rebuilt as well, and a new page will be built including the updated scoring information. Eventually, when the match completes, the complete page shown in the example is produced.

The score for the final match will be displayed on many pages other than Steffi Graf's player page. For instance, Martina Hingis's player page will also include these results, as will the scoreboard page while the match is in progress. A page listing matchups between different players will also contain the score. To update all of these pages, the automated system only updates one fragment. This keeps the automated system independent of the site design.

III. System Performance

This section describes the performance of a Java implementation of our system running on an IBM Intellistation containing a 333 Mhz Pentium II processor with 256 Mbytes of memory and the Windows NT (version 4.0) operating system. The distribution of Web pages sizes is similar to the one for the 1998 Olympic Games Web site [8]; the average Web page size is around 10 Kbytes. Fragment sizes are typically several hundred bytes but usually less than 1 Kbyte. They are representative of the fragments which will be used at the upcoming sports Web sites which will deploy our system in the near future.

Figure 9 shows the CPU time in milliseconds required for constructing and publishing bundles of various sizes. Times are averaged over 100 runs. All 100 runs were submitted simultaneously, so the times in the figure reflect the ability for the runs to be executed in parallel. The solid curve depicts times when all objects which need to be constructed are explicitly triggered. The dotted line depicts times when a single fragment which is included in multiple pages is triggered; the pages which need to be built as a result of the change to the fragment are determined from the ODG. Graph traversal algorithms applied to the ODG have relatively low overhead. By contrast, each object which is triggered has to be read from disk and parsed; these operations consume considerable CPU overhead. As the graph indicates, it is more desirable to trigger a few objects, which are included in multiple pages, than to trigger all objects which need to be constructed.

Our implementation allows multiple complex objects to be constructed in parallel. As a result, we are able to achieve near 100% CPU utilization, even when construction of an object was blocked due to I/O, by concurrently constructing other objects.

The breakdown as to where CPU time is consumed is

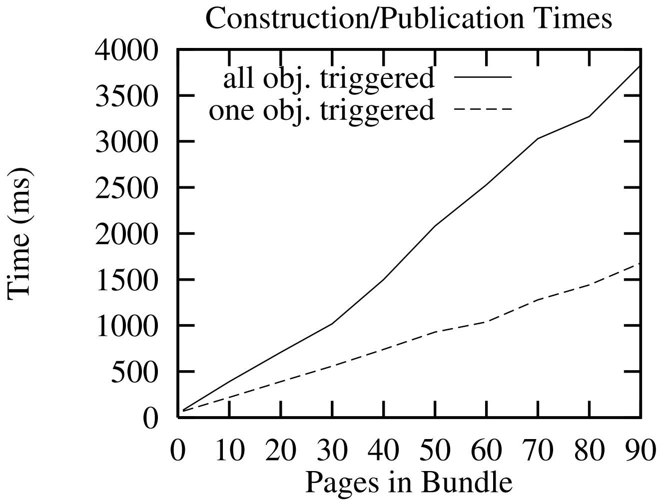

Fig. 9. The CPU time in milliseconds required to construct and publish bundles of various sizes.

shown in Figure 10. CPU time is divided into the following categories:

- *Retrieve, parse:* time to read all triggered objects from disk and parse them for determining included fragments.
- *ODG update:* time for updating the ODG based on the information obtained from parsing objects and for analyzing the ODG to determine all objects which need to be updated and an efficient order for updating the objects.
- *Assembly:* time to update all objects.
- *Save data:* time to save all updated objects on disk.
- *Send ack:* time to send an acknowledgment message via HTTP that publication is complete.

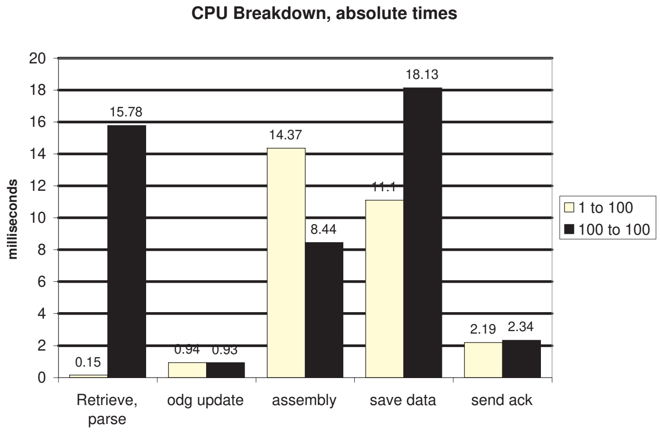

Fig. 10. The breakdown in CPU time required to construct and publish a typical complex Web page.

In the bars marked *1 to 100,* one fragment included in 100 others was triggered. The 100 pages which needed to be constructed were determined from the ODG. In the bars marked *100 to 100,* the 100 pages which needed to be constructed were all triggered. The times shown in Figure 10 are the average times for a single page. The total average time for constructing and publishing a page in the 1 to 100 page is 28.75 milliseconds (represented by the aggregate of all bars); the corresponding time for the 100 to 100 case is 45.62 milliseconds. Figure 11 shows the breakdown in CPU time when times are normalized so that the entire process (represented by the aggregate of all bars) consumes 1 time unit.

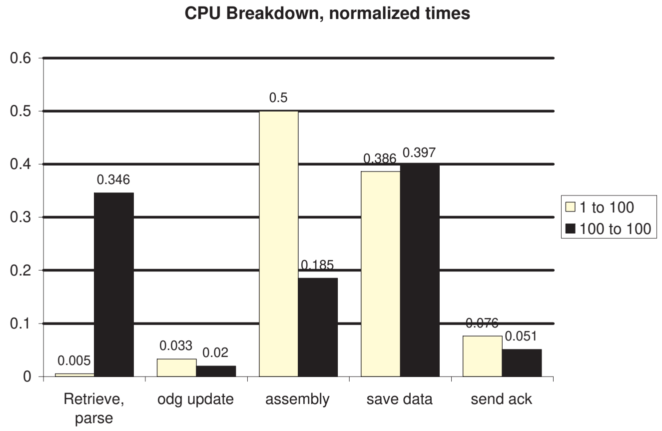

Fig. 11. The breakdown in normalized CPU time required to construct and publish a typical complex Web page.

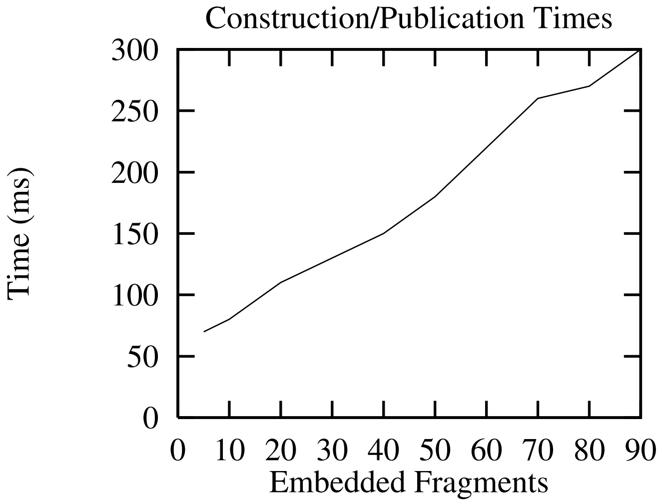

Fig. 12. The average CPU time in milliseconds required to construct and publish a complex Web page as a function of the number of embedded fragments. In each case, one fragment in the page was triggered.

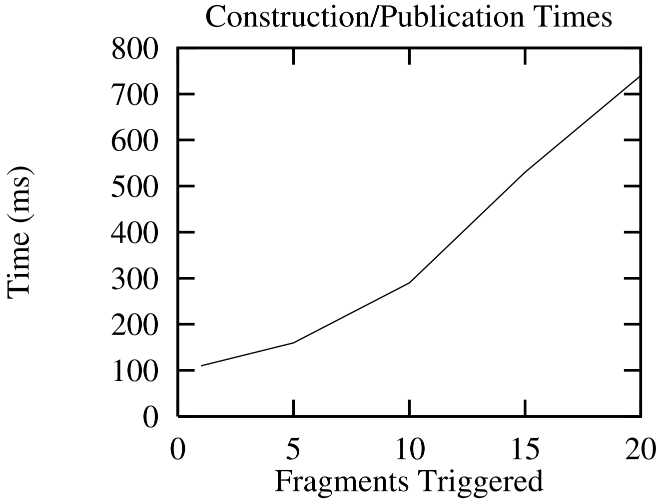

Fig. 13. The average CPU time in milliseconds required to construct and publish a complex Web page as a function of the number of fragments triggered.

The retrieve and parse time is significantly higher for the 100 to 100 case because the system is reading and parsing 100 objects compared with 1 in the 1 to 100 case. Assembly takes significantly longer in the 1 to 100 case because the system needs to read embedded fragments from disk for assembling each page. We are working on reducing assembly time by caching embedded fragments in memory.

Figure 12 shows how the average construction and publication time varies with the number of embedded fragments within a Web page. Figure 13 shows how the average construction and publication time varies with the number of fragments which are triggered for a Web page containing 20 fragments. Both graphs are averaged over 100 runs.

We are planning to deploy our system at a number of popular sports Web sites in the near future such as the ones for Wimbledon, the PGA, the Ryder Cup, the US Open tennis tournament, and the Sydney Olympic Games. For the final paper, we will present performance results from the upcoming sports events which will have elapsed by the time the final version is written.

IV. Related Work

There are a number of Web content management tools on the marketplace today such as NetObjects Fusion [11], Allaire's ColdFusion and Homesite [1], FutureTense's Internet Publishing System [5], Eventus Software's Control [6], Wallop Software's Build-It (now owned by IBM) [13], Site Technologies' SiteMaster [12], and Microsoft's Visual InterDev [10].

As far as we know, none of these products allow nested fragments to the degree which we do. Most of them don't allow any type of embedded fragments. They are also not designed to publish content in multiple stages as ours is.

A key problem with many products such as Fusion and SiteMaster is that they only work well when all of the Web content is designed using the product. They don't provide rich programmatic interfaces which can deal with or import content from external sources or feeds. These products thus lack the ability to treat external data with the same level of control and consistency as the sources of data the application owns.

By contrast, our system allows Web pages to come from multiple external sources. This is a key requirement for many of the Web sites we have encountered. Build-It is similar to our system in that it works with Web content created by other sources. However, we found that Build-It was not able to handle Web sites as large as the 1998 Olympic Games Web site, for example. Our system is scalable to handle extremely large Web sites.

Our system uses many ideas from the system used for the 1998 Olympic Games Web site [2], [3]. That system used an earlier version of the Trigger Monitor to maintain updated caches of dynamic data. The original Trigger Monitor maintained updated caches by reacting to database triggers. When a database change occurred, a database trigger invoked a UDF (User Defined Function) that sent a message to the Trigger Monitor containing an encoded summary of the change. The Trigger Monitor decoded the message, consulted an ODG to determine which pages were affected, requested pages from a non-caching HTTP server, and finally replaced the updated pages in the caches of the servers connected to the Web.

While the 1998 Olympic Games system worked extremely well for maintaining updated caches, it lacked the automated features of our new system for automatically and consistently publishing dynamic content. While the earlier system used object dependence graphs for determining how changes to underlying data affected cached objects, it didn't have capabilities for automatically constructing pages and fragments in an optimal order. The earlier system also couldn't publish combined content pages efficiently and had fewer options for bundling Web pages for consistent publication.

V. Summary and Conclusions

We have presented a publishing system for efficiently creating dynamic Web content. Our publishing system constructs complex objects from fragments which may recursively embed other fragments. Relationships between Web pages and fragments are represented by object dependence graphs. We presented algorithms for efficiently detecting and updating all affected Web pages after one or more fragments change.

After a set of multiple Web pages change or are created for the first time, the Web pages must be published to an audience. Publishing all changed Web pages in a single atomic action avoids consistency problems but may cause delays in publication, particularly if the newly constructed pages must be proofread before publication. Incremental publication can provide information faster but may also result in inconsistencies across published Web pages. We presented three algorithms for incremental publication designed to handle different consistency requirements.

Our publishing system provides an easy method for Web site designers to specify and modify inclusion relationships among Web pages and fragments. Users can update content on multiple Web pages by modifying a template. The system then automatically updates all Web pages affected by the change. It is easy to change the look and feel of an entire Web site as well as to consistently update common information on many Web pages.

Our system accommodates both quality controlled fragments that must be proofread before publication and are typically from humans as well as immediate fragments that have to be published immediately and are typically from automated feeds. A Web page can combine both quality controlled and immediate fragments and still be updated in a timely fashion.

Our publishing system has been implemented in both Java and C++. We will soon be deploying our system at several popular sports Web sites. The final version of the paper will describe our experiences with the actual deployments.

References

[1] Allaire's ColdFusion and HomeSite. http://www.allaire.com/.

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

[3] Jim Challenger, Arun Iyengar, and Paul Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE INFOCOM '99*, March 1999.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[5] FutureTense's Internet Publishing System. http://www.futuretense.com/.

[6] Eventus Software's Control. http://www.eventus.com/.

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

[8] Arun Iyengar, Mark Squillante, and Li Zhang. Analysis and Characterization of Large-scale Web Server Access Patterns and Performance. In *World Wide Web*, June 1999.

[9] Eric Levy-Abegnoli, Arun Iyengar, Junehwa Song and Daniel Dias. Design and Performance of a Web Server Accelerator. In *Proceedings of IEEE INFOCOM '99*, March 1999.

[10] Microsoft's Visual InterDev. http://www.microsoft.com/.

[11] NetObjects Fusion. http://www.netobjects.com/.

[12] Site Technologies' SiteMaster. http://www.sitetech.com/.

[13] Wallop Software's Build-It. http://www.wallop.com/.