Fine-Grained Parallelism in Dynamic Web Content Generation: The Parse & Dispatch Approach♦

Stavros Papastavrou[1], George Samaras[1], Paraskevas Evripidou[1], Panos K. Chrysanthis[2]

[1] Computer Science Department, University of Cyprus,
75 Kallipoleos St. P.O.Box 20537, Nicosia, Cyprus
{stavrosp, cssamara, skevos }@ucy.ac.cy
[2] Computer Science Department, University of Pittsburgh,
Sennott Square, Pittsburgh PA 15260, USA
panos@cs.pitt.edu

Abstract. Dynamic Web content is gaining in popularity over traditional static HTML as the means of providing Web users with personalized and dynamic information. To enable dynamic content, various technologies have been developed for embedding of script code blocks into static HTML files in order to perform various forms of tasks such as session tracking, bank transactions, financial calculations, products catalog generation, dynamic image generation, or even fetching information from remote servers. In this way, static HTML pages are transformed into dynamic web pages. Typically, dynamic Web pages include a number of tasks that are executed in a serial manner by current Web servers. In this paper, we propose a back-end, finer-grained parallel approach for dynamic content generation, and elaborate on how it affects the design and performance of Web servers. We have developed a prototype Web server that supports the parallel processing of tasks involved in the dynamic content generation with improved throughput as compared to the classical (serial) approach.

1 Introduction

Web servers are the basic component of the World Wide Web [4] in terms of content delivery. Early Web servers, such as the NSCA HTTPd Web server [15], were used for dissemination of static documents (files) based on a specification called the Hypertext Markup Language [13], a standard for publishing documents on the Internet. The need, however, for the delivery of non-static content, such as documents customized on the fly based on information provided by a single user, led to the specification of the Common Gateway Interface (CGI) [8, 21]. Web servers supporting the CGI generate dynamic content by running external programs (executables or scripts) that typically access an application specific database. According to CGI, each client request requires the separate execution of an external program.

♦ This work is partially funded by the Information Society Technologies programme of the European Commission under the IST-2001-32645 DBGlobe project.

With the great expansion of the Web, the need for a more scalable, persistent, and faster alternative to CGI revolutionized the design of Web servers. Modern Web servers [14] support multiple runtime environments in which various augmented versions of HTML code execute in a scripting mode in order to generate dynamic content. Such a runtime environment executes either as library code within the Web server process, or as a separate process communicating via a standard application interface. Microsoft's Active Server Pages [1], Macromedia's Cold Fusion [12], and PHP [19] are examples of such a runtime technology that support augmented versions of HTML.

A typical augmented HTML script file that is executed in order to generate a dynamic Web page consists of both standard HTML code and multiple insertions of vendor-specific script code. A piece of script code (code block) may map to a simple task, such as an animated counter or a personal menu bar generation, to complex tasks, such as lengthy distributed database transactions. As we explain later on, there can be notable delays in generating a dynamic Web page due to (a) the serial manner in which traditional Web servers execute those tasks, and (b) the long duration of executing a particular task.

In order to boost performance, recent studies propose a number of content-aware and link-aware dispatching and load balancing algorithms to be used over a cluster of Web servers [3, 5, 10, 11, 20]. According to *content-aware dispatching*, HTTP requests are routed to specialized Web servers based on their content type whereas *link-aware dispatching* routes HTTP request by mapping their URL using a hash table. Both approaches can boost the total throughput of a Web site's hosting environment by independently executing entire HTML scripts in parallel, however, they do not improve the throughput of a single Web server per se.

In this paper, we focus on improving a Web server's performance in producing dynamic Web content by introducing parallelism at finer granularities. In nutshell, our idea is to execute the tasks included in a dynamic Web page in parallel based on the proposed *Parse and Dispatch* approach. Our goal is not limited only in accelerating Web server performance, but also in identifying the design principles and limitations of such an approach. We study a novel Web server design and compare its performance to that of a traditional Web server that executes those tasks in a serial manner. To the best of our knowledge, there is neither related literature nor a Web server product that employs such a design. Furthermore, we study the design and performance challenges for our approach in the presence of dependencies between tasks included in a dynamic Web page.

As opposed to *Proxy-based* [7] and server-side caching [6] approaches, *Parse and Dispatch* is introduced in this paper as a *back-end* Web server *acceleration* technique, in the sense that it is employed at the opposite side from the Web client, i.e., at the other end of the client/server communication path. The path may also include a client cache, a proxy server caching system(s), and a Web server cache plug-in mechanism.

The rest of this paper is organized as follows: In Section 2, we discuss some necessary background information on dynamic content and how Web servers generate and deliver it. In Section 3, we introduce and analyze our methodology for dynamic content generation, while in Section 4 we put our methodology to the test assuming no dependencies among tasks. However, in some cases, dependencies may exist between

two (or more) tasks included in an HTML script file. For example, the total amount owed by a visitor in a retail Web site cannot be generated unless her shopping cart has been validated. In Section 5, we present one way to handle dependencies using our approach. We conclude and discuss future work in Section 6.

2 Background Information

2.1 Dynamic Content

Modern Web sites utilize dynamic content technology in order to respond with dynamic and personalized content to clients/visitors without having to construct and store it a-priori. Thus, it is possible to tailor the content served to a client according to her most recent needs and expectations, while saving huge amounts of disk space. Web applications that are benefited by dynamic content technology include e-commerce, online financial brokers, portals and news related sites.

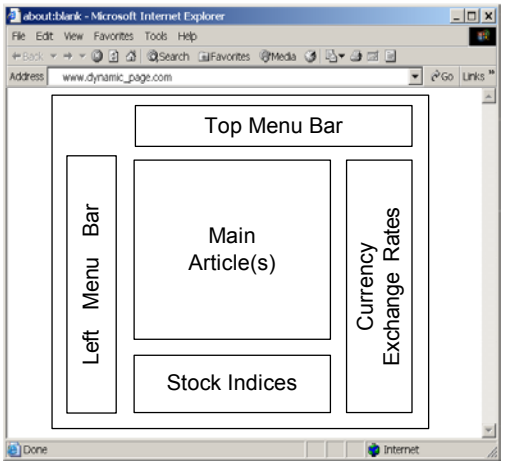

Figure 1: The layout of a typical stock-related dynamic Web page

A typical dynamic Web page includes tasks that are processed at the Web server site in order to produce the dynamic fragments of the Web page. Although the number of tasks may vary across Web pages of different applications, we assume that this number lies between four and eight. For instance, consider a typical stock-related dynamic page (Figure 1) with the following tasks:

- Session handling (not seen in Figure 1).
- Left menu bar generation. Links are dynamically generated from the local content database.

- Top menu bar generation. Links are customized to a particular client's preferences stored in the local content database.
- One or more main articles/stories retrieval. Articles are extracted from a corporate news database.
- Current stock market indices retrieval. Indices are pulled from a remote financial provider via XML, and they can be presented either graphically or in text.
- Currency exchange rates retrieval (Similar to stock market indices).

In addition, e-commerce dynamic Web pages include tasks such as catalog generation code, user cart handling code, credit card verification, banner advertisements rotation, counter update and more. Product customization pages (i.e., from a computer retailer site) include tasks that consist of thousands of lines of script code and multiple database queries in order to generate dynamic Web forms for product customization.

Dynamic Web pages are generated with the parsing of static files (HTML script files) normally located in the file system of the Web server but not necessarily under a public directory. Such files may have extensions such as ".asp", ".cfm", or ".php" that denote different scripting languages from various vendors. The asp extension stands for Active Server Pages, a technology developed by Microsoft Corp. that supports the insertion of Visual Basic code (vbscript) blocks that may generate dynamic content. Cold Fusion, a product of Macromedia Inc., uses a tagged-based script code while PHP, o project of Apache Software Foundation, supports a Unix-like script code.

In any case, we can safely assume that the static content (HTML) and dynamic content (blocks of script code) are both arranged in HTML script files in an interleaved manner. Static content is transmitted to the Web user as is. The blocks of script code that perform the tasks related to the dynamic Web page, however, are substituted by their execution output which is then transmitted to the Web user.

This interleaved usage of static and dynamic content is a popular way of defining the layout/arrangement of the dynamic parts in a dynamic Web page. For example, the <table> tag, and the <tr> and <td> sub tags, are used to define the placement of the dynamic content under the assumption that a Web page's layout can be simply overviewed as a grid. Consequently, HTML script files are often called "HTML templates" and are widely employed in Content Management Tools.

In past few years, Content Management Systems (CMS) have been gaining in popularity as the means for serving and managing dynamic Web content. Such tools provide a secured graphical user interface for the Web site administrator to update the contents of the Web site by updating the site's database. To materialize a dynamic Web page, the CMS parses an HTML template given that page's name or unique identifier. The http://host.com/generatepage.asp?pagename=home URL, for example, will generate on the fly the home page of the host.com site.

2.2 Dynamic Content Generation

In this subsection, we illustrate in brief the traditional processing steps taken by a Web server in order to generate dynamic content. We assume the usage of a Multi-Threaded Web server over Event-Driven or Single-Process Web servers (for a descriptive

comparison between Web server architectures, please see [18]). This assumption is based on the fact that popular multi-threaded Web servers, such as Microsoft's Internet Information Server [16] and Apache's HTTP Server [9], generate the substantial majority of dynamic Web content today. For a detailed report on Web server usage, please see [17].

Figure 2 highlights the structure and functionality of a multi-threaded Web server. A single parent process accepts incoming HTTP requests from Web clients in a sequential manner through a server socket. Upon arrival of a new request, an available worker thread is selected out of a pool of suspended pre-dispatched threads in order to serve that request. Meanwhile, the parent process continues execution free to accept more requests, enabling in this way the concurrent processing of multiple clients. Once a worker thread has served its assigned request, it places itself back to the thread pool.

An upper limit on the number of worker threads allowed to execute in parallel is set in order to ensure the proper (non-thrashing) execution of the Web server. In popular Web servers, the pre-configured size of the thread pool ranges from five to twenty. In the case that all worker threads are busy, excessive client requests are not accepted, however, they are queued in the special buffer called *backlog* for upcoming admission. With the backlog buffer full, additional client requests are refused.

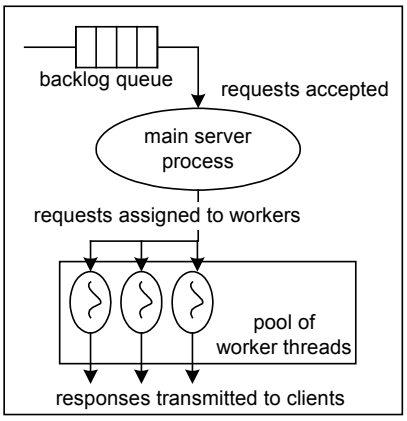

Figure 2: A multi-threaded Web server

Client requests for static content refer to files located in the file system of the Web server under a pubic html directory. Static HTML files usually have a ".html" or "htm" extension and contain only standard HTML code. Following a client request for a static content, a worker thread extracts the file name from the client's HTTP request header, and searches for that file, first, in the Web server's cache, and then in the file system. Once the file is found, an HTTP response header is sent to the client followed by the contents of the requested file. The contents of a static HTML file are not always sent in one chunk. A repeat-until loop loads and transmits successive fragments of the file until the end of the file is reached.

As mentioned above, client requests for dynamic content refer to HTML script files (usually having an asp, cfm, or php extension). Following a client request, the worker

thread assigned to the particular client must locate the appropriate file and, according to the file's extension, invoke a handler method from the appropriate library provided by the corresponding vendor.

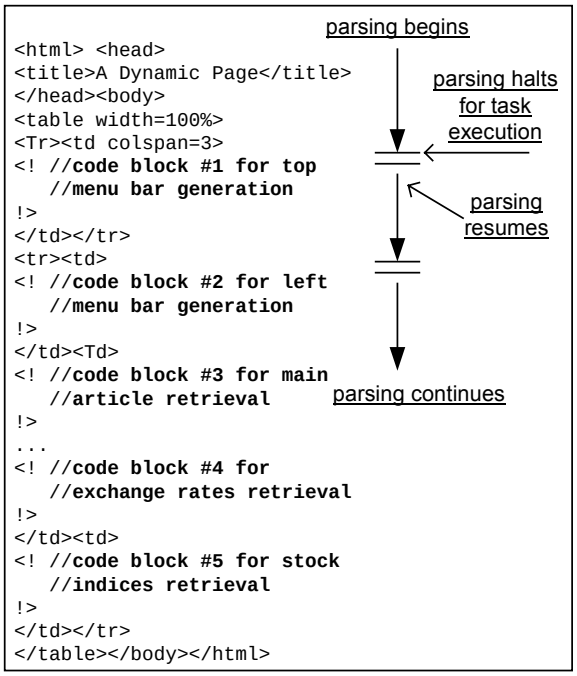

Figure 3: Contents and parsing of file MainPage.dyn

Running within the worker thread's resources, the handler method opens and parses the script file. Static HTML content is appended in a temporary buffer while script code forces the handler method to *pause parsing*, for as long as it takes, in order to execute the task. The script code output (in HTML) is appended in the buffer (buffered mode) and the handler method continues parsing. With the end of file reached, the entire contents of the buffer are transmitted to the client following an HTTP response header. Some scripting languages explicitly allow the transmission of content as soon as this becomes available (unbuffered mode), however, a comparison in performance and network utilization between buffered and unbuffered content transmissions on Web servers remains an open topic for research. In either case, blocking the parsing of the HTML script file for task execution directly hurts performance since it creates an unnecessary processing bottleneck.

Figure 3 demonstrates the processing steps required to generate a sample home page by executing the MainPage.dyn[1] HTML script file that includes five tasks. The vertical arrows, pointing downward, represent the parsing steps and the horizontal lines the

[1] We use our own file extension (.dyn), which stands for 'dynamic', for HTML script files in favor of no particular vendor.

temporal halting of the parsing in order to execute the encountered script code. The generated content is rendered by the client's browser to display a top menu bar, a left menu bar, the main article of the day followed the current exchange rates, and finally the current stock marker indices. The two menu bars are generated by running queries on a local (to the Web server) database provided the personal preferences of the user. The main article is loaded from a news database, while the current exchange rates and stock indices are acquired from a remote financial provider via XML.

In order to generate the dynamic content of MainPage.dyn, the parsing of the file was suspended for five times. Table 1 contains an approximation of the execution time for various tasks that scripting code might refer to. The execution time of tasks depends on various parameters such as computational complexity, hardware power and network speed.

Table 1: Approximate execution time for various tasks

Task	Execution Time
Counter Update	~ 10 ms
Session Handling	~ 10 ms
SQL execution	10 to 100 ms (depending on complexity)
Image Generation	Multiples of 10 ms (depending or resolution and contents)
Remote data retrieval	Multiples of 100 ms (depending on connectivity)

3 A new Model for Dynamic Content Generation

With both the popularity of dynamic content and number of Web users growing, a more efficient processing methodology is needed for materializing dynamic Web pages. The current serial processing manner in which the tasks of a dynamic Web page are executed by traditional Web servers is computationally and implementation-wise simple, yet it is not efficient. We realize efficiency in terms of computational resources utilization that is translated into improved Web server throughput.

Our suggested methodology puts more parallelism in dynamic content generation by processing the tasks, embedded in a dynamic HTML file, in a concurrent fashion based on the proposed *Parse and Dispatch* approach. Our methodology provides an additional level of parallelism under the one obtained by using clustered Web servers. We next present our approach that assumes no dependencies between tasks.

3.1 The Parse and Dispatch Approach without Dependencies

The intuition behind the Parse and Dispatch approach is to enable the uninterrupted/non-blocking parsing of an HTML script file by assigning the execution of the script code blocks (tasks) to auxiliary threads that run in parallel. The approach

consists of two phases: (a) the Content Expansion Phase, and (b) the Content Serialization Phase.

The Content Expansion Phase. Following a client request, the Web server selects an available worker thread out of the thread pool that will carry out the request. The worker thread initializes an indexed buffer (a variable-length array with variable-length strings as elements) that will be used as a temporary content storage. With the current index at the buffer set to 1, the worker thread locates the requested HTML script file and opens it for parsing.

The first consecutive block of static content (see Figure 4) is stored as a character string at buffer index 1 and the current index is increased by one. Then, the thread worker detects the first block of script code and initializes an auxiliary thread that will execute the code. Reserving the current index on the buffer for the auxiliary thread, the worker thread increases the current index by one and continues parsing.

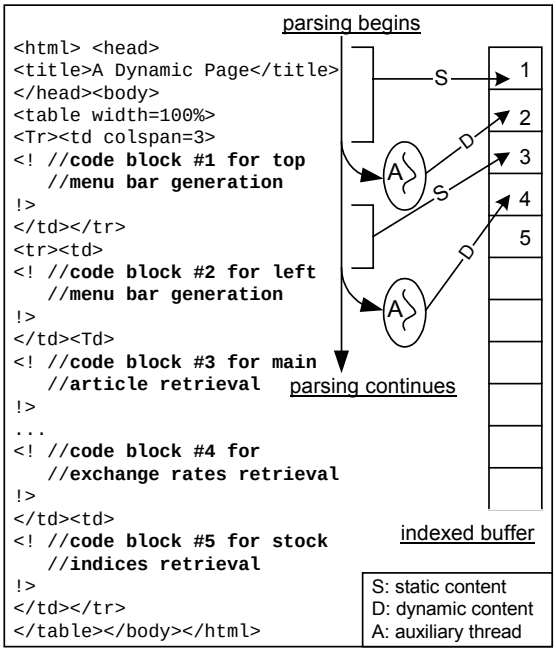

Figure 4: Contents and parsing of file MainPage.dyn using the Parse and Dispatch approach

The same tactic is followed for the rest of the file contents. In case of having two consecutive blocks of script code (with no static content separating them), two successive buffer indices will be reserved. A block of uninterrupted static content is never split. The content expansion phase ends as soon as all tasks are under processing.

The Content Serialization Phase. An auxiliary thread that finishes its script code execution, stores the generated content in the buffer position that was reserved for it during the content expansion phase and terminates. The worker thread waits for all the dynamic content to become available. (In our current implementation, the worker thread detects the termination of all the auxiliary threads by periodically checking the reserved buffer indices.) With all the auxiliary threads terminated and the dynamic content available, the worker thread scans the buffer and transmits both static and dynamic content to the client. Scanning the buffer from index 1 and up ensures that the content parts are delivered to the client in the right order.

Figure 4 illustrates both the expansion phase and the usage the indexed buffer. The vertical arrow, pointing downward, represents the file parsing, while a curved arrow represents the concurrent execution of the encountered script code by an auxiliary thread. The "S" arrows denote the placement of static content in the buffer, and the "D" arrows the placement of the dynamic content in the buffer. Figure 5 displays the a processing timeline of the approach.

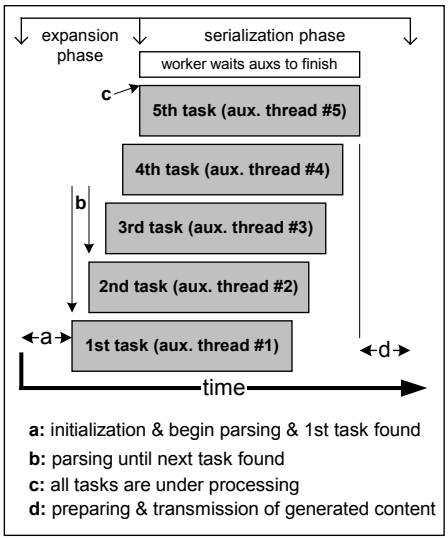

Figure 5: Processing timeline of file MainPage.dyn using the Parse and Dispatch approach

3.2 Improving the Content Serialization Phase

During the content serialization phase, the worker thread makes sure that all the auxiliary threads have stopped executing by periodically checking whether they have placed their generated content in their reserved buffer index. In order to minimize the overhead of such a procedure, the ideal scenario would be that all the auxiliary threads terminate as soon as the worker thread finishes parsing and enters the content serialization phase.

The worker thread, however, enters the content serialization phase before all of the auxiliary threads have terminated and wastes valuable computational resources waiting for them (point c in Figure 5). We can exploit those wasted resources, by assigning to the worker thread itself the execution of the last script code block of the HTML script file. The benefits of such an optimization are twofold. First, we shorten the period that the worker thread spends waiting for the auxiliary threads to terminate. Second, we decrease by one the number of auxiliary threads initialized during the expansion phase. Thread initialization may not be as expensive as process forking, nevertheless, it poses a significant overhead in a computationally intensive, multithreaded application such as a Web server. Figure 6 displays the processing timeline with the above improvement.

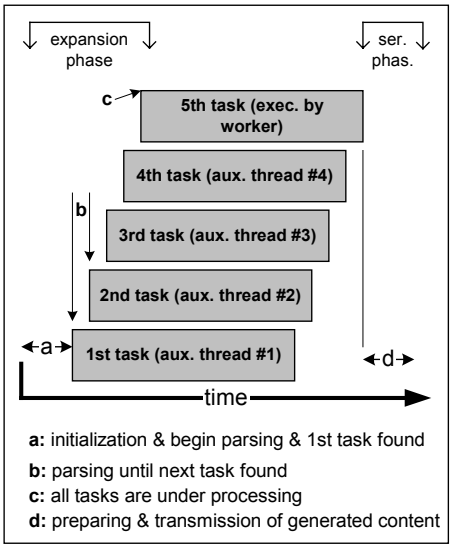

Figure 6. Processing timeline of MainPage.dyn using the improved Parse & Dispatch approach

The challenge is how the worker thread detects the final script code block in order to hold back from initializing the last auxiliary thread. An apparent solution would be to delay an auxiliary thread initialization and parse ahead to detect the next script code block occurrence or detect the end of file. Nonetheless, such a tactic would harm performance since it widens the time gap between the initialization of auxiliary threads and thus reducing parallelism. A more efficient solution requires from the worker thread to know, prior to the expansion phase, the exact number of script code blocks in an HTML script file. For that reason, an additional piece of code is inserted in the beginning of a file in the form of a pre-processor directive. Such a directive may look like "<script_blocks count=5/>", or "<tasks count=5/>" and it is the first code that the worker thread parses during the Content Expansion phase.

4 Evaluating the Parse and Dispatch Approach without Dependencies

In our experiments, we compare the performance between (a) a traditional multi-threaded Web server that executes the tasks of dynamic Web page in serial (as described in Section 2.1), and (b) an experimental multi-threaded Web server that executes the tasks of a dynamic Web page in parallel according to the Parse and Dispatch approach assuming no dependencies between two tasks. Next, we describe our experimental setup in terms of hardware, software and topology. We then discuss the experiments and our findings.

4.1 Experimental Setup

For the development of the traditional Web server, we adopted the thread management and client admittance routines from the Java-based Apache Web server (Jserv). We then developed the proposed experimental Web server by (a) modifying the code of the worker threads and, (b) adding support for the auxiliary threads and the indexed buffer. We chose to derive the experimental Web server from the traditional one to ensure maximum compatibility between the performance results of the two approaches. The decision for using Java was based on the language's rich, easy-to-use APIs that speed-up the programming of experimental multi-threaded, and network applications [2].

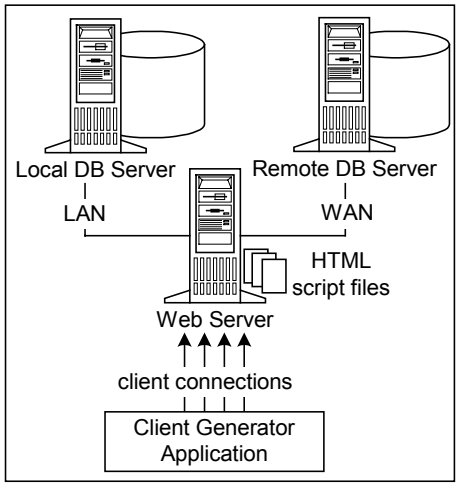

Figure 7: Topology of the experiments

For both the traditional and the experimental Web servers, we assume a worker thread pool of size 21, meaning that both Web servers do not admit more than 21 concurrent clients. We base our assumption on the fact that popular Web servers

recommend a size between 5 and 20. It will be made clear by the performance results that this assumption does provide a complete scope of results for evaluating the two approaches.

Our next consideration is the formation of the HTML script files that the Web servers will process in the experiments, given that dynamic Web pages from various applications differ on the blend and number of tasks that they include. For example, a financial-related Web page may include more tasks that generate dynamic content with data obtained by remote financial providers. On the other hand, an e-commerce Web page may include more tasks for dynamic catalog generation by querying local databases.

For the structure of the tasks, we assume a typical task with an approximate execution time of 50 milliseconds that favors no particular type of application. The typical task consists of the following: (a) script code for two queries performed randomly either on a local or a distant database to emulate local or remote database access, (b) script code for string manipulation that executes in long loops to emulate dynamic HTML generation. The decision on forming the typical task is based on certain experience gained in working with commercial Web sites.

The topology of the experimental setup is shown in Figure 7 and attempts to emulate a real-world commercial environment. The Web server (either the traditional or the experimental one) runs on our main server machine in our local area network. The HTML script files to be processed are copied under the Web server's public directory. The first content database server is installed locally to the Web server while the second one is installed on a remote machine through a WAN.

The last piece of our setup is the application program that implements the client requests called the Client Generator. This application is capable of instantiating a predefined number of individual client programs each one capable of independently submitting consecutive HTTP requests to a Web server. Client programs are implemented by threads and are arbitrated, similar to a Web server, by the Client Generator program using a thread pool. The Client Generator program can emulate a number of concurrent clients that submit requests on a Web server for a given period. To avoid interfering with the Web server's computational resources, the Client Generator resides locally on a different machine.

All the machines used in the experiments were Pentium 4 class computers with 1GB of main memory and SCSI-based secondary storage. The local network was a 100Mps Ethernet, and the remote network a 1Mbps WAN. The database servers used were both Microsoft SQL Server 2000 and the Java Virtual Machine for both the Web servers was version 1.3.1.

4.2 Experiments and Performance Results

We measure the performance of a Web server in terms of (a) average throughput, and (b) client response time under different (stable) workloads. Formally, we define throughput to be the average number of client HTTP requests that are completely processed by a Web server in a period of one second and it is computed at the Web server site. We define client response time to be the client perceived latency from the

moment the client issues an HTTP request to the Web server, until the client receives the complete HTTP response.

In our experiments, Web server workload depends on the number of client requests that are simultaneously admitted by the Web server for processing. We also refer to this as 'Concurrent Clients'. For our experiments, workload varies from 1 to 21 since a Web server with a worker thread pool of size 21 does not admit more than 21 concurrent clients. Due to space shortage, in this paper we present our findings for 2 and 5 embedded tasks in an HTML script file.

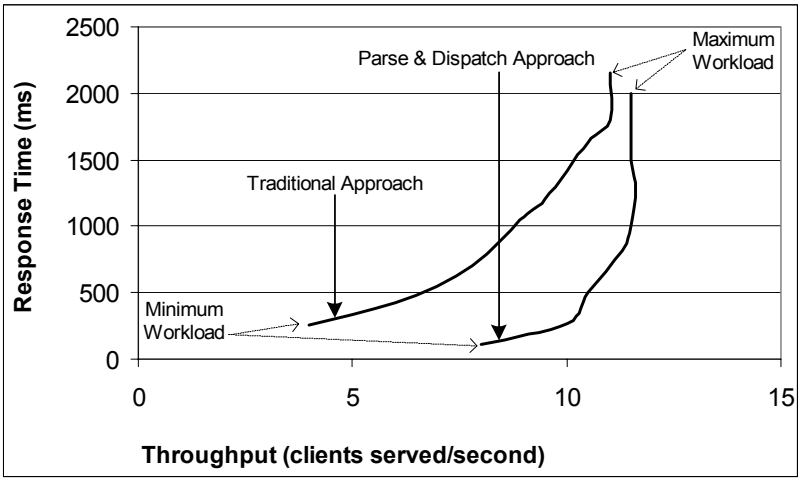

Figure 8: Performance Comparison with 5 tasks included in an HTML script file with no dependencies between tasks

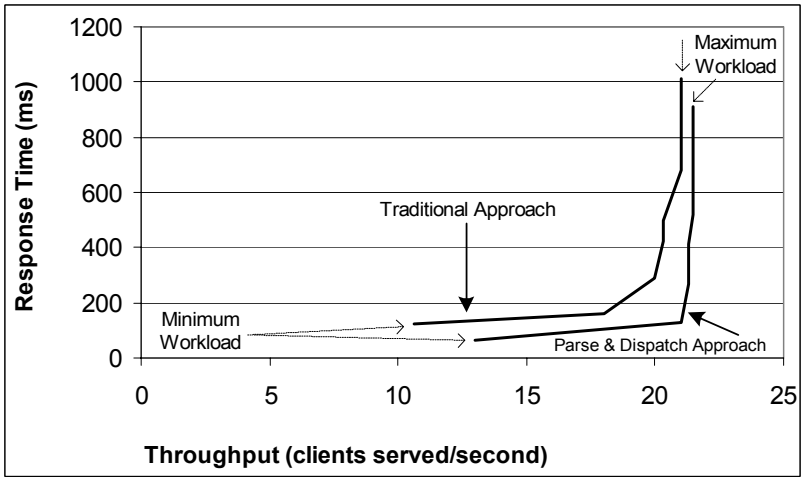

Figure 9: Performance Comparison with 2 tasks included in an HTML script file with no dependencies between tasks

The results shown in Figure 8 indicate that, for five embedded tasks in an HTML file, the suggested Parse and Dispatch approach outperforms the traditional approach in both response time and throughput. However, the performance gains for the Parse and Dispatch approach are less obvious for two included tasks (Figure 9). This is because by having only two tasks in an HTML file, we reduce the amount of parallelism that the Parse and Dispatch approach can benefit from.

5 Parse & Dispatch Approach with Dependent Tasks

In some Web applications, dependencies may exist between two or more tasks included in the some HMTL script file. For example, a graph (jpeg image) that compares two stock quotes cannot be generated unless the two user-supplied quote names have been successfully retrieved, and analyzed. Since traditional Web servers execute the tasks of an HTML page in serial, additional special handling of tasks is not required (given that the developer of the Web page has put the tasks in the right order!). In the case of our approach, we had to enhance our task dispatching algorithms.

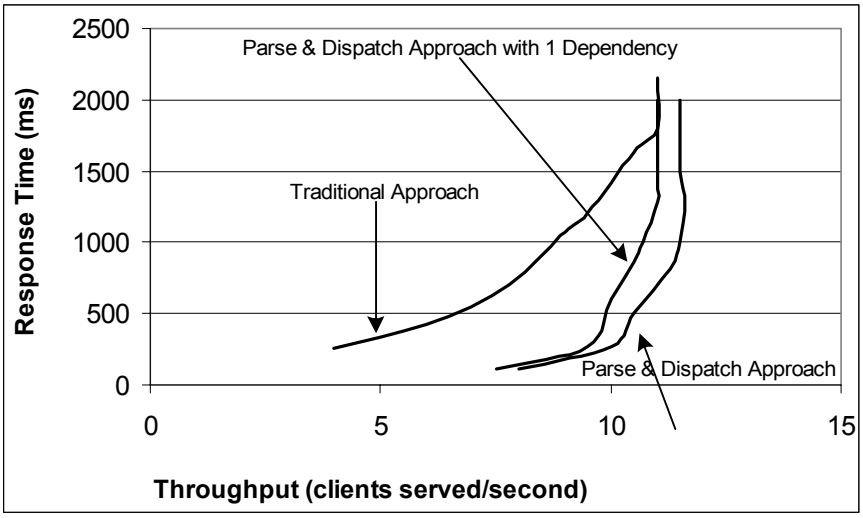

Figure10: Average throughput with 5 tasks included in an HTML script file plus the average throughput of the Parse & Dispatch approach having 2 dependent tasks (1 dependency)

In doing so, we came across two challenges. First, we had to find a way of letting the Parse & Dispatch Web server know about the dependencies. Then, we had to implement a cost-efficient mechanism of executing the dependent tasks in serial. For the former, we use a pre-processor directive of the form <dependency source=task1 target=task3>, where *source* and *target* denote the numbering order of two dependent

tasks in the HMTL script file. For the latter, our experiments showed that it is more efficient to have the auxiliary thread that executed the source task to also execute the destination task. In this way, (a) we prevent the dispatching of one additional auxiliary thread, and (b) we minimize the time gap between the successive executions of the dependent tasks.

Figure 10 shows the performance results of the Parse & Dispatch approach with five included tasks and having two dependent tasks (1 dependency) next to our earlier results reported in Figure 8 with five included tasks but with no dependencies. For this experiment, we have used various orderings for the dependent tasks and calculated their average performance. The results indicate that the performance gains still hold even at the existence of one dependency between two tasks.

6 Conclusions and Future Work

In this paper, we introduced a new approach for dynamic Web content generation. Our approach suggests parallelism at the granularity of dynamic Web page fragments, in addition to that of a whole Web page obtained by using clustered Web servers. The proposed approach, Parse and Dispatch, was used to build an experimental Web server and its performance was compared to that of a traditional Web server. The experimental results yielded significant performance gains in favor of our approach in terms of Web server throughput and client response time.

As part of our future work, we are further developing a more efficient Web server architecture focused on dynamic content, and particularly pay attention on applying parallel processing techniques in handling dependencies between tasks included in a HTML script file.

References

1. Active Server Pages. Available at http://www.microsoft.com/
2. K. Arnold, J. Gosling: The Java Programming Language. Addison-Wesley 1996.
3. J. Aweya, L M. Ouellette, D. Y. Montuno, B. Doray, K. Felske. An adaptive load balancing scheme for web servers. Int. J. Network Mgmt 2002; 12: 3 – 39.
4. T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, A. Secret: The World-Wide Web. CACM 37(8): 76-82(1994).
5. V. Cardellini, E. Casalicchio, M. Colajanni, P. S. Yu: The state of the art in locally distributed Web-server systems. ACM Computing Surveys 34(2): 263-311 (2002).
6. A. Datta, K. Dutta, K. Ramamritham, H. M. Thomas, D. E. VanderMeer: Dynamic Content Acceleration: A Caching Solution to Enable Scalable Dynamic Web Page Generation. SIGMOD Conference 2001.
7. A. Datta, K. Dutta, H. M. Thomas, D. E. VanderMeer, Suresha, K. Ramamritham: Proxy-based acceleration of dynamically generated content on the world wide web: an approach and implementation. SIGMOD Conference 2002: 97-108.
8. G. Ehmayer, G. Kappel, S. Reich: Connecting Databases to the Web: A Taxonomy of Gateways. DEXA 1997: 1-15.

9. R. T. Fielding, G. E. Kaiser: The Apache HTTP Server Project. IEEE Internet Computing 1(4): 88-90 (1997).
10. X Gan, T. Schroeder, S. Goddard, B. Ramamurthy: LSMAC and LSNAT: Two Approaches for Cluster-Based Scalable Web Servers. ICC (2) 2000: 1164-1168
11. S. Goddard, T. Schroeder: The SASHA Architecture for Network-Clustered Web Servers. HASE 2001: 163-172
12. G. Hutchinson, G. Baur, D. Pigford: Implementation of a Dynamic Web Database: Interface Using Cold Fusion. SIGUCCS 1998: 131-135
13. HyperText Markup Language (HTML). Overview available at http://www.w3.org/MarkUp.
14. K. Kant, P. Mohapatra: Workshop on Performance and Architecture of Web Servers (PAWS-2000, held in conjunction with SIGMETRICS-2000). SIGMOD Record 29(3): 12-14 (2000).
15. E. D. Katz, M. Butler, R. McGrath: A Scalable HTTP Server: The NCSA Prototype. Computer Networks and ISDN Systems 27(2): 155-164 (1994).
16. Microsoft Internet Information Server. Available at http://www.microsoft.com.
17. Netcraft Web Server Survey. Available at http://www.netcraft.com/survey/.
18. V. Pai , P. Druschel, W. Zwaenepoel: Flash: An Efficient and Portable Web Server. Proceedings of the 1999 USENIX Annual Technical Conference, Monterey, CA, USA, June 1999.
19. PHP. Available at http://www.php.net/.
20. M. Di Santo, F. Frattolillo, W. Russo, E. Zimeo: Efficient Content-aware Connections Dispatching in Clustered Web Servers. PDPTA 2002: 843-849.
21. The Common Gateway Interface. Overview available at http://hoohoo.ncsa.uiuc.edu/cgi/overview.html.