Efficient Resource Management on Template-based Web Servers

Eli Courtwright   Chuan Yue   Haining Wang
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23187, USA
{eli,cyue,hnw}@cs.wm.edu

Abstract

The most commonly used request processing model in multithreaded web servers is thread-per-request, in which an individual thread is bound to serve each web request. However, with the prevalence of using template techniques for generating dynamic contents in modern web servers, this conventional request processing model lags behind and cannot provide efficient resource management support for template-based web applications. More precisely, although content code and presentation code of a template-based dynamic web page can be separated into different files, they are still processed by the same thread. As a result, web server resources, especially database connection resources, cannot be efficiently shared and utilized. In this paper, we propose a new request scheduling method, in which a single web request is served by different threads in multiple thread pools for parsing request headers, performing database queries, and rendering templates. The proposed scheme ensures the high utilization of the precious database connections, while templates are being rendered or static contents are being served. We implemented the proposed scheme in CherryPy, a representative template-enabled multithreaded web server, and we evaluated its performance using the standard TPC-W benchmark implemented with the Django web templates. Our evaluation demonstrates that the proposed scheme reduces the average response times of most web pages by two orders of magnitude and increases the overall web server throughput by 31.3% under heavy loads.

Keywords: Web Server, Resource Management, Templates, Performance, Request Scheduling.

1 Introduction

There are two important trends in modern web application development. One is the use of templates to dynamically generate HTML web pages, and the other is the practice of storing a database connection in each web server thread. We highlight both trends as follows.

- One of the main web design principles is to separate content code from presentation code. This is one of the primary design motivations behind Cascading Style Sheets (CSS) [14], which let web designers separate the display from the web content and specify how a web page should be displayed in a file. Now many templating languages exist and allow web authors to write HTML files with special tags. These tags are replaced at runtime when a template is rendered with data, which is often pulled from a database.
- Connections to such a database are often stored in the web server's threads with two purposes. First, this eliminates the overhead of establishing a new database connection every time when a page is loaded. Second, this keeps a programmer from having to close or free up each connection in the code for each page, which is often troublesome when dealing with multiple execution paths and thrown exceptions.

These two trends have become more apparent in modern web sites with the increasing need for providing the efficient and mass generation of dynamic web pages. Unfortunately, the current request processing model in multithreaded web servers does not provide adequate resource management support to these technical trends. The major problem is that modern multithreaded web servers are still using the traditional thread-per-request model to process requests for template-based web applications. Thus, for a dynamic web page, although its content code and presentation code are separated into different files, these files are still processed by the same thread for a specific request. Consequently, precious database connection resources cannot be efficiently shared and utilized, because they cannot be used by other threads to prepare data even if their holding threads are rendering templates and do not need the database connections at that time.

In this paper, we propose a new request scheduling method under the consideration of both trends to greatly improve multithreaded web server performance. In the proposed method, a web server uses different threads in multiple thread pools for parsing request headers, generating data, and rendering templates. By assigning database connections only to data generation threads, we ensure that these connections do not sit idle while templates are being rendered or static contents are being served. This separation of request services into different thread pools makes the database much less of a bottleneck. Without separation, if each thread has its own connection, then the number of threads cannot exceed the number of connections. Thus, a request might wait for a thread loading static content or rendering a template to finish before it can query the database. Alternatively, if every single thread becomes tied up performing lengthy database queries, then requests for static content will have to wait for those queries to finish.

Our request scheduling method alleviates the problems above even further by having different pools of threads for serving quick and lengthy dynamic requests. This prevents short database queries from being blocked by large ones and thus achieves effects similar to Shortest Job First scheduling, but without causing the starvation of lengthy jobs. In addition, when measuring the time each page takes to generate data from database, the time it takes to render templates will not be included because template rendering is now handled by a separate pool of threads. This increased accuracy of execution time measurement, in turn, helps us perform better request scheduling and resource management.

To validate the efficacy of the proposed request scheduling method, we implemented it into CherryPy [15], a representative template-enabled multi-threaded web server, and we implemented the standard TPC-W [24] benchmark in Django template language [16]. Our evaluation demonstrates that the proposed request scheduling method can reduce the average response times of most web pages by two orders of magnitude and increase the overall web server throughput by 31.3% under heavy loads.

The remainder of this paper is structured as follows. Section 2 introduces the background of modern web application techniques. Section 3 details the proposed request scheduling method. Section 4 presents the evaluation results. Section 5 discusses related work on request scheduling. Finally, Section 6 concludes the paper.

2 Background

While the single-process event-driven architecture can provide excellent performance for cached workloads and static web contents, the multithreaded architecture usually performs better for disk-bound workloads and dynamically generated web contents. Currently most popular and commercial web sites use multithreaded servers to provide web services.

```
<%
pageid = FieldStorage(req).get("pageid")
dbconn = connect("localhost", "username",
                 "password", "db")
cursor = dbconn.cursor()
cursor.execute("SELECT title, heading FROM page
                WHERE pageid=%s", pageid)
title, heading = cursor.fetchone()
%>

<html>
<head> <title> <%= title %> </title> </head>
<body>
<h2 align="center"> <%= heading %> </h2>
<ul>
<%
cursor.execute("SELECT data FROM sometable
                WHERE pageid=%s", pageid)
for row in cursor:
    req.write("<li>" + row[0] + "</li>")
%>
</ul>
</body>
</html>

<%
cursor.close()
dbconn.close()
%>
```

Figure 1. Traditional Python Server Page.

In this section, we detail two issues that motivate our work: (1) the advantages of the web template model over the traditional dynamic content generation model, and (2) the drawbacks of the conventional request processing model in multithreaded web servers.

2.1 Using Web Templates

The following example illustrates the difference between using web template and traditional web programming. Figure 1 shows the traditional way of generating dynamic web content. The code is written as a Python Server Page, which allows us to embed Python code in HTML. This approach mixes data generation and presentation code. Consequently, a large project coded in this manner forces a programmer to search through presentation code to find something related to data generation, or vice versa. Unfortunately, this is the traditional way to write code using techniques such as JSP [20], PHP [21], and ASP [23].

In contrast, using a modern web server such as CherryPy [15] and a templating language such as Django [16] allows us to separate our content from presentation code. Thus, we can write a function that performs the database queries of the above code and then renders a template using that data.

```
def example(self, pageid):
 data = {}
 cursor = getconn().cursor()
 cursor.execute("SELECT title, heading FROM page
                 WHERE pageid=%s", pageid)
 data["title"], data["heading"] = cursor.fetchone()
 cursor.execute("SELECT data FROM sometable
                 WHERE pageid=%s", pageid)
 data["listitems"] = [row[0] for row in cursor]
 cursor.close()
 return get_template("tmpl.html").render(Context(data))
```

Figure 2. Data preparation function in Django.

This function can also retrieve the existing database connection from its web server thread, instead of opening and closing a new one on every request. Figures 2 and 3 show the data preparation function and the presentation template written in Django template language, respectively.

```
<html>
<head> <title> {{ title }} </title> </head>
<body>
<h2 align="center"> {{ heading }} </h2>
<ul>
{% for item in listitems %}
    <li> {{ item }} </li>
{% endfor %}
</ul>
</body>
</html>
```

Figure 3. Presentation template (tmpl.html) in Django.

The template itself is mostly simple HTML with just a few special tags, which indicate how to render the template with the given data. While very simple, the example should give readers who are unfamiliar with modern templating languages a good understanding of what they are and how they are used.

2.2 Thread-per-request Model

The most commonly used request processing model in multithreaded web servers is the thread-per-request model. In this model as shown in Figure 4, an incoming request is first accepted by the single listener thread. Then, the request will be dispatched to a separate thread in the thread pool, which processes the entire request and returns a result to the client. To avoid the overuse of resources, the size of the thread pool is often bounded in most web servers, and meanwhile, a limited number of database connections are stored and shared by the threads.

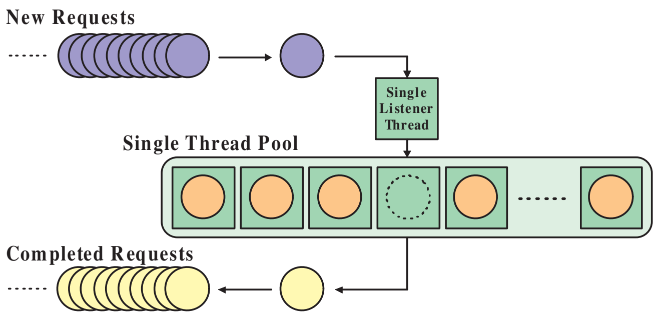


Figure 4. Thread-per-request model.

This model increases concurrency, improves performance, and is easy for programming. Unfortunately, it does not effectively support the recent trend of using web templates to separate content code from presentation code. More precisely, although content code and presentation code can be separated into different files using modern template techniques as shown in Figures 2 and 3, they are still executed by the same thread for each specific request. Thus, the precious database connection resources will be wasted by those threads that are rendering the presentation code while still holding database connections.

3 Design

In this section, we first introduce the necessary modification on web templates to support our new request scheduling method. Then, we detail the design of thread pools and present the request scheduling policy. We use CherryPy as an example of modern template-enabled multithreaded web servers and use Django as an example of modern web templates to illustrate our design.

3.1 Modification to Web Templates

CherryPy is an object-oriented HTTP framework written in Python, and it is designed to facilitate the development of dynamic web applications. It conveniently maps URLs to functions, converting each request's query string into function parameters. This allows developers to write web code in basically the same way as they would write code for conventional applications.

In order to support the proposed request scheduling method, a minor modification on a web application's templates is needed. The modification does not significantly change the way that CherryPy code is written. CherryPy programmers write a function for each dynamic page, generate data in that function, and return a rendered template, as shown in Figure 2. The only difference between this normal procedure and our modified version is that instead of returning a rendered template, each function returns an unrendered template and the rendering data. Using the Django template code as an example, the conventional way of rendering templates is:

return get_template("tmpl.html").render(Context(data)),

where *tmpl.html* is the name of the template file and *data* is the dictionary (a.k.a. hashtable) used to render the template. This conventional return statement will thus return a string that contains a rendered dynamic web page. By contrast, our modified version of rendering templates is:

return ("tmpl.html", data),

and thus this new return statement simply returns the name of the unrendered template and the data to be rendered later. That is the only change made on web templates.

Such a minor modification maintains the consistency of the way template code is written, which is very important for the wide use of the proposed scheme. If a new web server dramatically changes how programmers write code, they will be reluctant to adopt it. This is why our modification requires only the return statement of each function to be different. For example, in our Django implementation of TPC-W benchmark, only 14 lines of return statements (one for each type of dynamic pages) need to be changed in order for the whole benchmark to take advantage of the performance benefits offered by the proposed request scheduling method. Moreover, even if a function returns an already-rendered template by mistake, the modified web server can still handle this properly although it cannot apply our proposed request scheduling method for rendering the template in a different thread.

3.2 Thread Pools

Like most multithreaded web servers, CherryPy also uses the common thread-per-request model as shown in Figure 4 to process client requests. It has a single listener thread which accepts incoming TCP connections and places each of them in a synchronized queue. A large pool of threads waits on that queue, and once a connection is available, a thread takes it from the queue and services the entire request before waiting on the queue again.

The key feature of our request scheduling method is to use multiple thread pools rather than just one thread pool to serve different web requests. Our new model has a listener thread with five different thread pools: Header Parsing, Static Requests, General Dynamic Requests, Lengthy Dynamic Requests, and Template Rendering. Each thread pool waits on its own synchronized queue. An incoming request is first accepted by the single listener thread, and then passed to the five thread pools for processing. The processing flows are shown in Figure 5.

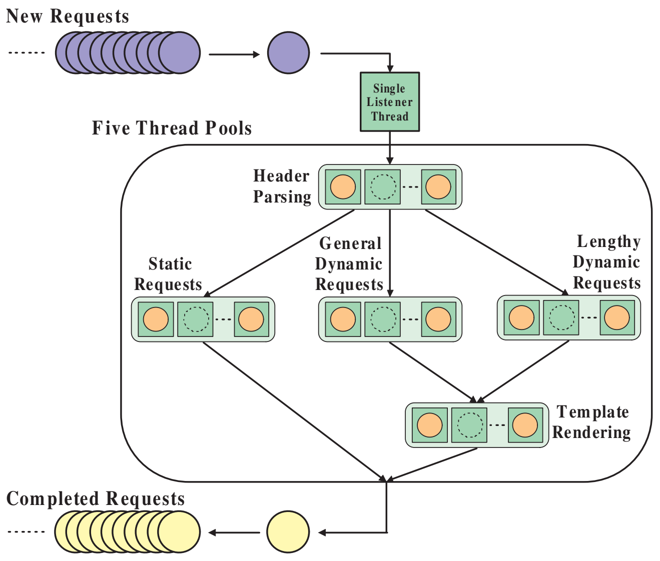


Figure 5. Thread pools on the modified web server.

The header parsing threads parse the first line of each HTTP request. The first line contains the path of the resource being requested, which is critical to tell whether that resource is a static file or a dynamically generated page. Each request is then placed into either the static request thread pool or one of the two dynamic request thread pools. For example, one way to distinguish between static and dynamic requests is to check the extension of the requested resource. Suppose that a header parsing thread reads the line:

```
GET /img/flowers.gif HTTP/1.1
```

then the thread can examine the request and know that it is requesting a file which ends in ".gif"— a static file. The header parsing thread thus dispatches the request to the static request thread pool. However, if the request is instead:

```
GET /homepage?userid=5&popups=no HTTP/1.1
```

then the header parsing thread can check to ensure that the resource "/homepage" does not have any kind of extension and thus is a dynamic resource. Therefore, the header parsing thread dispatches the request to one of the two dynamic request pools. The criteria used for determining which dynamic request thread pool serves the request are explained in Section 3.3.

If the request is for a dynamic page, the header parsing thread also parses the rest of the HTTP request's header data. In our dynamic request example above, suppose that the entire request is:

```
GET /homepage?userid=5&popups=no HTTP/1.1
User-Agent: Mozilla/1.7
Accept: text/html
......
```

then in addition to placing the request in one of the dynamic request queues, the header parsing thread will further parse the query string "userid=5&popups=no" and the two headers "User-Agent" and "Accept". The headers and query string will each be parsed into a dictionary (a.k.a. hashtable). We perform these further parsings mainly because we do not want a thread with an open database connection to waste time doing anything other than generating data. This is not an issue for static requests, so we let the threads which actually serve those static requests parse their headers.

Each dynamic request thread maps the request string to a function, then examines the function's return value to see whether it is a string or a template to be rendered. Every function should return an unrendered template as described in Section 3.1, and we perform this check to allow backward compatibility so that unmodified template code can still properly run on our modified web server. If the function returns a string, then the dynamic request thread directly sends the string to the client. If the function returns a template, then the dynamic request thread passes the request on to the pool of template rendering threads.

After one of the template rendering threads finishes the rendering of a dynamic page, it measures the size of the output. Thus, it is able to set the Content-Length HTTP response header appropriately, which cannot be achieved by most existing methods in dynamic content generation. The template rendering thread then transmits the response to the user agent client.

3.3 Scheduling Policy

Our request scheduling method uses two thread pools for serving dynamic requests: a general dynamic request thread pool, and a lengthy dynamic request thread pool. We refer these two pools as the general pool and the lengthy pool, respectively. The lengthy pool only handles the requests which take a long time to serve. We used a cutoff point of two seconds to distinguish between quick and lengthy requests, which is suitable for our benchmark.

The general pool handles both quick and lengthy dynamic requests, and the requests which can be processed quickly are always served by this pool. Because our main priority is to ensure that quick requests do not get stuck in a queue behind a number of lengthy requests, the general pool has four times as many threads as the lengthy pool.

In order to distinguish between quick and lengthy requests, we track the average time spent in generating data for each page. Specifically, we measure the time cost in the dynamic request thread, from when the request is acquired through when its unrendered template is placed in the template rendering queue. This gives us the accurate measurement of how much time is spent in performing database queries.

condition	**dispatch decision**
a quick request	send to general pool
a lengthy request and *tspare* $>$ *treserve*	send to general pool
a lengthy request and *tspare* $\leq$ *treserve*	send to lengthy pool

Table 1. Dynamic request dispatching rules.

To ensure that quick requests are almost always served immediately, our web server tracks the number of spare threads in the general pool, which we call *tspare*. It also keeps updating a shifting minimum number of threads reserved for quick requests, called *treserve*. The *tspare* is a measured value that reflects the load of the web server, while the *treserve* is a dynamically adjusted value that reflects the targeted number of threads that should be reserved for quick requests. In the case of *tspare* being greater than *treserve*, the spare threads in the general pool is abundant and they can serve some lengthy requests in addition to processing all the quick requests. In the case of *tspare* being no greater than *treserve*, the spare threads in the general pool fall short and should be dedicated to serving quick requests. Therefore, when a header parsing thread receives a lengthy request, it sends the request to the general pool if *tspare* is greater than *treserve*; otherwise the request is sent to the lengthy pool. Table 1 summarizes the three rules for dispatching a dynamic request by a header parsing thread.

Our web server checks and modifies *treserve* once per second in order to deal with traffic spikes. Because we want to prevent quick requests from being queued behind lengthy requests, we increase *treserve* anytime we suspect that a traffic spike is occurring. Specifically, whenever *tspare* drops under *treserve*, we increase *treserve* by the difference, plus the amount that *tspare* has dropped beneath a configured minimum value of *treserve*, if applicable.

We lower *treserve* more slowly to avoid prematurely assuming that a traffic spike has ended. When *tspare* rises above *treserve*, we lower *treserve* by half the difference but without making it less than the configured minimum value. Table 2 lists the dynamics of *treserve* vs. *tspare* over a 10-second period with the minimum value of *treserve* configured as 20. Intuitively, whenever a traffic spike is occurring and spare threads in the general pool become scarce, we increase the *treserve* so that more lengthy dynamic requests can be dispatched to the lengthy pool, thus reserving the threads in the general pool only for quick requests. By contrast, when a traffic spike tends to disappear and the spare threads in the general pool become abundant, we decrease the *treserve* so that these spare threads can also be used to process incoming lengthy dynamic requests.

time	tspare	treserve	Δtreserve
1s	35	20	+0
2s	24	20	+0
3s	17	20	+6
4s	21	26	+5
5s	30	31	+1
6s	36	32	-2
7s	38	30	-4
8s	37	26	-5
9s	35	21	-1
10s	39	20	+0

Table 2. Changes to *treserve* over an example 10-second period.

Note that while we have multiple thread pools for serving dynamic requests, we only have one thread pool for static requests and one thread pool for template rendering. This is because the time difference in serving different static requests or rendering different templates is much less significant than that caused by different requests which require database queries. However, applying this technique to static requests and template rendering might be worthwhile on a different benchmark or web application.

4 System Evaluation

In this section, we first describe the experimental setup including the TPC-W benchmark implemented with Django web templates, and the testbed configuration. Then we present the experimental results.

4.1 Experimental Setup

We employ TPC-W, a transactional web e-commerce benchmark [24] for the performance evaluation of the proposed request scheduling method. TPC-W exercises an online bookstore, which supports a full range of activities, such as multiple on-line sessions, dynamic page generation, and online transactions. It also specifies a workload generator that simulates many clients visiting a web site. This benchmark is well designed to mimic a typical real world web application, particularly by having database accesses be the bottleneck when the site comes under heavy load.

Unfortunately, existing TPC-W implementations are all written using the traditional techniques for dynamic web content generation. For example, PHARM team of the University of Wisconsin - Madison [17] developed Java servlets implementation of TPC-W benchmark, and a team of Rice University [18] developed PHP, Java Servlets, and EJB implementations of TPC-W benchmark. Therefore, we have to implement TPC-W benchmark from scratch using a modern web template technique. We choose Django template, which is supported by CherryPy web server and is used by the Washington Post for much of its online content. Our implementation of the TPC-W benchmark consists of 455 lines of Python code and 704 lines of template code (most of which is pure HTML). It's worth noting that this implementation has less than 1/4 as many lines of code as the Java Servlets implementation by PHARM team.

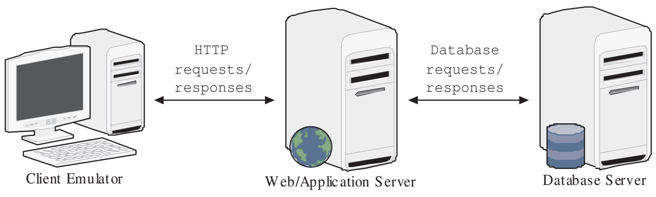


Figure 6. Experimental testbed.

In our experimental testbed shown in Figure 6, three computers are used to perform the TPC-W benchmark. One computer is used to run the MySQL 5.0 database server, one is used as the CherryPy web server to host both the static and dynamic content, and one is used as the workload generator. Each computer has 8 different 3.7GHz CPUs, each with 16KB of L1 cache and 2MB of L2 cache. They each have 8GB of RAM and run Linux with kernel version 2.6.18, and are all connected to the same local area network with a 100Mb network interface card. The duration of each experiment is one hour and the measurement interval is 50 minutes. The first five-minute ramp up time and the last five-minute cool down time are not included.

The TPC-W database is configured to have one million books, 2.88 million customers, and 2.59 million book orders. All experimental runs described in this paper are conducted with standard "browsing mix" workload, and each simulated client waits the standard time of 0.7 to 7 seconds before visiting a new page. The workload generator simulates 400 clients in order to put the web server under a heavy load.

The servers are set up as in a production environment, where the web and database servers would certainly be on the same LAN. Of course, most clients would not be connected from the same network, so the transfer times for this benchmark are far below what they would be for a real web site. However, this should not be a problem here because we are primarily interested in the decrease of database query response times rather than transfer latencies.

4.2 Experimental Results

We use two performance metrics to compare our proposed scheme with the traditional thread-per-request scheme: web interaction response time and web server throughput. In TPC-W, a web interaction refers to a complete cycle of the communication between an emulated client and the server system under test. The web interaction response time is measured at the client-side by calculating the time lapsed from the first byte of a web interaction request sent out by a client to the last byte of the web interaction response received by the client. The web server throughput is measured at the server-side in terms of the number of completed web interactions per minute.

In the rest of this section, we use the term "unmodified web server" to refer to the conventional thread-per-request CherryPy web server, on which TPC-W benchmark with unmodified Django web templates is run. We use the term "modified web server" to refer to our proposed multiple-thread-pool CherryPy web server, on which TPC-W benchmark with modified Django web templates is run.

4.2.1 Web Interaction Response Time

The average response times of running the benchmark with the unmodified and modified web servers are listed in Table 3. For 11 out of the 14 pages of the benchmark web site, the proposed scheme significantly shortens the web interaction response times. The average response times of many pages like the homepage (TPC-W_home_interaction) are decreased by two orders of magnitude. One slow page (TPC-W_new_products) stays about the same, one (TPC-W_execute_search) becomes slightly slower to respond, and one (TPC-W_admin_response) is clearly taken longer time to respond.

The sharp performance dichotomy between the extremely fast and very slow pages in the modified web server is partially due to the TPC-W benchmark itself. Most of the queries are either select statements making use of an index, or insert statements adding a new row. Neither of these operations are very slow even on extremely large databases; creating a database with 10 times the size of the current one does not cause the fast queries to become noticeably slower.

Of the 14 pages in the TPC-W benchmark, 10 are inherently very fast (less than 10 seconds) for the reasons described above, three are very slow because they perform large and very complex queries, and the last one is very slow because it performs an update on a frequently used table. This last page is the TPC-W_admin_response page, which is the only page to experience a significant slowdown with our modified web server. In order to perform its update, it must acquire a lock on a database table, forcing it to wait for other threads to finish the use of the table. Ironically, this page is slower to respond for our modified server because the other pages are so much more efficient, lengthening the wait time to acquire the table lock. In fact, in the absence of any considerable load on the server, this page is quite fast.

web page name	unmodified	modified
TPC-W_admin_request	4.89	0.62
TPC-W_admin_response	12.35	18.85
TPC-W_best_sellers	18.49	12.88
TPC-W_buy_confirm	3.86	0.18
TPC-W_buy_request	3.74	0.07
TPC-W_customer_registration	4.46	0.01
TPC-W_execute_search	11.05	13.21
TPC-W_home_interaction	2.54	0.03
TPC-W_new_products	20.30	21.39
TPC-W_order_display	2.78	0.54
TPC-W_order_inquiry	4.84	0.04
TPC-W_product_detail	1.10	0.01
TPC-W_search_request	5.44	0.01
TPC-W_shopping_cart_interaction	6.82	0.27

Table 3. TPC-W pages and their average response times (seconds) on the unmodified and modified web servers.

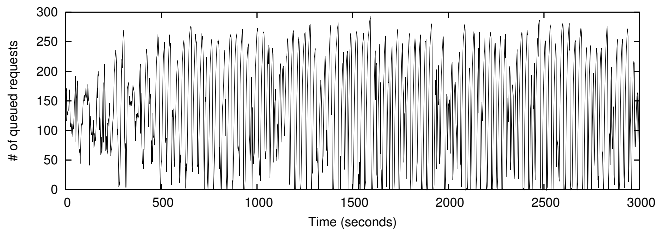
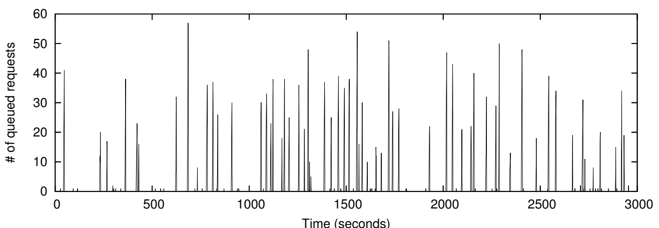


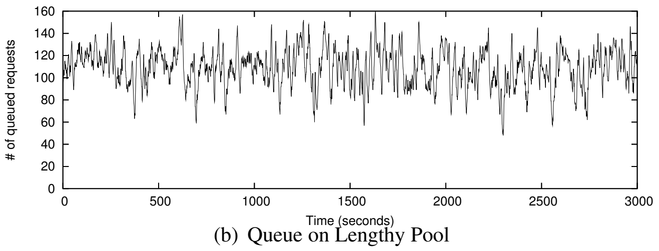
Figure 7. Length of the queue for dynamic requests on the unmodified web server.

A slow admin page is not as troublesome as a slow page visible to customers on an e-commerce site. While paying customers might leave a site which is slow in response, this problem does not exist for the pages only visible to admins. Thus, the mostly suffered page in the modified server is the page that matters least to the profitability of an online bookstore.

Figure 7 illustrates the effect that these slow queries have on the response times of other web pages by showing the length of the request queue for the unmodified web server. It is clear that the queue length tends to be very large when short requests get stuck behind lengthy requests in the queue. As comparison, Figures 8(a) and 8(b) show the length of queues that are associated with the two dynamic request thread pools in the modified web server. On the one hand, the short queries are able to execute almost immediately because there are threads reserved for them in the general dynamic request thread pool. On the other



(a) Queue on General Pool



(b) Queue on Lengthy Pool

Figure 8. Length of the queues for dynamic requests on the modified web server. (a) General; (b) Lengthy.

hand, this is also why we cannot obtain the same large performance gains for the lengthy requests as we do for the quick requests. Many of the lengthy requests get stuck in their own queue behind a number of other lengthy requests. Lowering the number of threads set aside for the quick requests improves the response time of lengthy requests, but at the cost of sacrificing much of the performance gains for those quick requests, particularly during traffic spikes. This would not be an acceptable tradeoff for a real web site.

While the response times for slow web pages are still high, on a real web site there are a number of things we could do to mitigate this. For example, we could add indexes to all fields in order to prevent queries from having to scan through an entire table to find fields with certain values. We could also regularly precompute expensive queries which do not change from user to user, such as which books are new or best sellers. Of course, to do these would change the TPC-W benchmark itself, and would deviate from the purpose of using such a standardized benchmark.

4.2.2 Web Server Throughput

Figure 9 illustrates the overall throughputs for all types of requests on the unmodified and modified web servers. It is clear that our proposed scheme consistently performs better than the traditional thread-per-request scheme. Figure 10 shows the detailed throughput comparison for four types of requests: static requests, all dynamic requests, quick dynamic requests, and lengthy dynamic requests. We can see that the throughput gains are obvious for all the four types of requests.

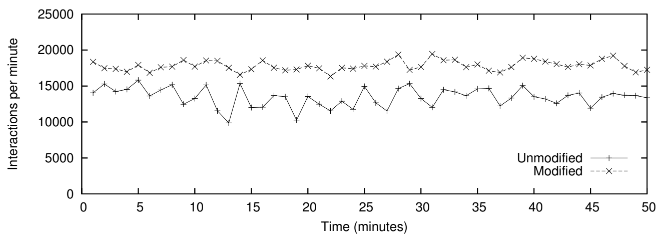
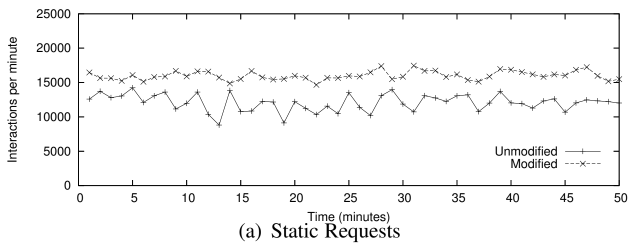


Figure 9. Throughput for all types of requests on the unmodified and modified web servers.

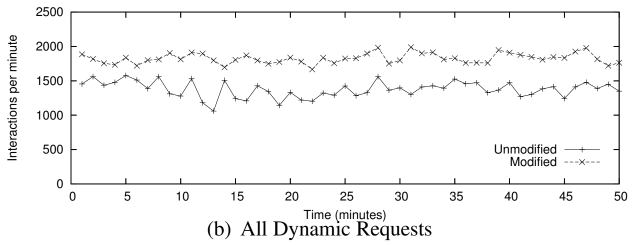
web page name	unmodified	modified
TPC-W_admin_request	74	81
TPC-W_admin_response	71	72
TPC-W_best_sellers	7602	9646
TPC-W_buy_confirm	395	547
TPC-W_buy_request	429	596
TPC-W_customer_registration	469	642
TPC-W_execute_search	7307	9723
TPC-W_home_interaction	19586	25608
TPC-W_new_products	7406	9758
TPC-W_order_display	184	206
TPC-W_order_inquiry	219	255
TPC-W_product_detail	14002	18608
TPC-W_search_request	7994	10543
TPC-W_shopping_cart_interaction	1173	1536

Table 4. The total numbers of completed web interactions for each type of TPC-W pages on unmodified and modified web servers.

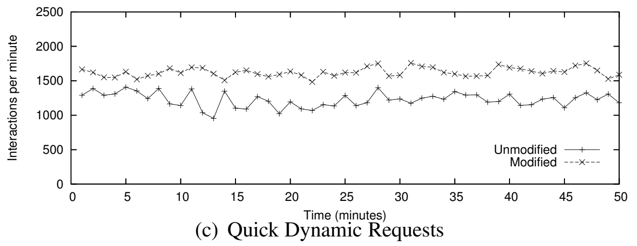
Table 4 further lists the total number of completed web interactions for each type of TPC-W pages during the 50-minute measurement interval. Obviously, by improving the utilization of database connections and enabling better scheduling, our scheme can increase the throughput of each type of web interactions. Overall, our scheme improves web server throughput over the traditional thread-per-request scheme by 31.3%. Note that for some requests, there is no strong correlation between the response time results (Table 3) and the throughput results (Table 4), while one expects them to follow Little's law. For instance, in case of the TPC_admin_response page, the average response time is increased by 52.6% in the modified web server (row 2 of Table 3), but the total number of completed web interactions remains almost the same (row 2 of Table 4). We conjecture that this irregularity is mainly due to the fact that TPC-W benchmark is a closed queuing system. Meanwhile, since the standard "browsing mix" workload of TPC-W is used in our experiments, the response time is not necessarily inversely proportional to throughput for each type of requests.



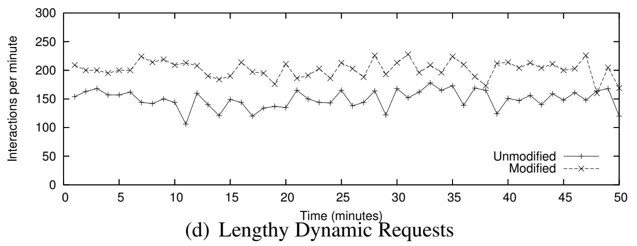
(a) Static Requests



(b) All Dynamic Requests



(c) Quick Dynamic Requests



(d) Lengthy Dynamic Requests

Figure 10. Throughput for each type of requests on the unmodified and modified web servers.

5 Related Work

Welsh and Culler [11] proposed a web server with a similar concept to ours. Each type of action such as querying databases, accessing files, and serving static contents would be handled by a different pool of threads. However, their focus is on load shedding at each stage rather than mitigating database latency. More important, they proposed to change the way that web developers write web applications, which severely limits the adoption of their approach. The main goal of our work is to achieve comparable results while keeping the programming model intact.

Considering both the size of a request and how long the request has been queued, Cherkasova [3] proposed a solution to overcome the shortcomings of Shortest Job First scheduling in web servers. She demonstrated that the proposed scheduling method can improve the average response time per HTTP request more than three times under heavy loads. Our approach makes a similar tradeoff among multiple queues without actually using any kind of priority queue.

Our scheme is also closely related to the work done by Elnikety *et al.* [5], which performs scheduling directly to database queries. Our scheme accomplishes much the same thing inside the web server and does not require an additional server to sit between the web and database servers. Thus, their work is more transparent to the actual application but requires more effort and hardware support to set up on the part of the site administrators.

In [6], Fowler *et al.* grew and shrunk thread pools to respond to demand. Their approach is complementary to ours. While we concentrate on using different thread pools to lower response times and increase web server throughputs, they adjusted thread pools to seek the optimal levels for different server loads. Indeed, they also used the TPC-W benchmark to demonstrate the effectiveness of their approach.

Crovella *et al.* [4] proposed shortest-connection-first scheduling to approximate the effects of Shortest Remaining Processing Time (SRPT) scheduling. Schroeder and Harchol-Balter [9] provided a detailed implementation study on SRPT scheduling and demonstrated its effectiveness in improving the performance of web servers during transient periods of overload. However, SRPT is more applicable to web servers that serve static content, i.e., files whose size can be determined in advance. In contrast, our method is a generic approach that specifically considers the trend of generating dynamic web pages with templates in modern web applications.

Some other work employ intelligent queuing and load balancing to improve web server performance, albeit with slightly different goals. Guitart *et al.* [7] proposed a session-based adaptive overload control mechanism to prioritize requests from existing SSL sessions rather than prioritizing small requests, because of the high overhead in setting up an SSL session. This work is complementary to ours, as it could be integrated with our queuing strategy if we run our benchmark over HTTPS. In [2], Bhoj *et al.* used a queuing strategy to improve quality of service for its more important clients. Each request is given a priority based on the user's classification rather than any particular property of the request itself. In [1], Abdelzaher and Bhatti proposed a web server QoS management architecture that relies on web content adaptation. Such an adaptation architecture enables a server to cope with overload in a graceful manner. Ranjan *et al.* [8] proposed to perform priority scheduling to mitigate distributed denial of service attacks. Instead of scheduling requests based on the estimated service time, they prioritized the requests which are more likely to originate from legitimate clients. Totok and Karamcheti [10] proposed a reward-driven request prioritization mechanism. This mechanism predicts the future structure of web sessions and gives higher execution priority to the requests whose sessions are likely to bring more reward.

In comparison with these previous work, our approach is novel because it separates template rendering from data generation, which significantly decreases the response latency and increases the accuracy of execution time measurement. Our request scheduling method also has a much higher chance of gaining widespread adoption than many of previous solutions, as it has little to no effect on how programmers write code. As mentioned at the beginning of the paper, separating content and presentation code has become common in the field of web programming. Similar to the Django framework, other modern web programming frameworks such as the Apache Struts [12], the ASP.NET MVC [13], the JavaServer Faces [19], and the Ruby on Rails [22] all separate their program logic from their HTML generation. It is this separation that enables the optimizations presented in the paper; therefore, our request scheduling method could be applied to each of those frameworks as well.

6 Conclusion

This paper presents a new request scheduling method to efficiently support the resource management of modern template-based web applications in multithreaded web servers. In this method, a web request is served by different threads in multiple thread pools for parsing headers, performing database queries, and rendering templates. By assigning database connections only to data generation threads, the proposed scheme ensures that open database connections spend less time idle, and hence, improves the utilization of the precious database connection resources. Meanwhile, using different threads increases the measurement accuracy of the service time for each type of requests, and thus enable us to provide better scheduling for dynamic web requests. We implemented our request scheduling method in CherryPy, a representative template-enabled multithreaded web server, and we evaluated its performance using the standard TPC-W benchmark implemented with Django web templates. Our evaluation results demonstrate that the proposed scheme reduces the average response times of most web pages by two orders of magnitude and increases the overall web server throughput by 31.3% under heavy loads.

Acknowledgments: We thank anonymous reviewers for their valuable comments and suggestions. This work was partially supported by NSF grants CNS-0627339 and CNS-0627340.

References

[1] T. F. Abdelzaher and N. Bhatti. Web server qos management by adaptive content delivery. In *Proc. of the IWQoS*, 1999.

[2] P. Bhoj, S. Ramanathan, and S. Singhal. Web2K: Bringing QoS to Web Servers. Technical Report HPL-2000-61, HP Laboratories, May 2000.

[3] L. Cherkasova. Scheduling strategy to improve response time for web applications. In *Proc. of the HPCN Europe*, pages 305–314, 1998.

[4] M. E. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proc. of the USITS*, pages 22–22, 1999.

[5] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proc. of the International World Wide Web Conference*, 2004.

[6] R. Fowler, A. Cox, S. Elnikety, and W. Zwaenepoel. Using performance reflection in systems software. In *Proc. of the HotOS*, 2003.

[7] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguade. Session-based adaptive overload control for secure dynamic web applications. In *Proc. of the ICPP*, 2005.

[8] S. Ranjan, R. Swaminathan, M. Uysal, and E. W. Knightly. Ddos-resilient scheduling to counter application layer attacks under imperfect detection. In *Proc. of the INFOCOM*, 2006.

[9] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Inter. Tech.*, 6(1):20–52, 2006.

[10] A. Totok and V. Karamcheti. Improving performance of internet services through reward-driven request prioritization. In *Proc. of the IWQoS*, 2006.

[11] M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In *Proc. of the USITS*, 2003.

[12] Apache Struts. http://struts.apache.org/.

[13] ASP.NET MVC. http://www.asp.net/mvc/.

[14] Cascading Style Sheets. http://www.w3.org/Style/CSS/.

[15] CherryPy - Trac. http://www.cherrypy.org/.

[16] Django — The Web framework for perfectionists with deadlines. http://www.djangoproject.com/.

[17] Java TPC-W Implementation Distribution by PHARM team. http://www.ece.wisc.edu/ pharm/.

[18] Java TPC-W Implementation Distribution by Rice University. http://www.cs.rice.edu/CS/Systems/DynaServer.

[19] JavaServer Faces Technology. http://java.sun.com/javaee/javaserverfaces/.

[20] JavaServer Pages Technology. http://java.sun.com/products/jsp/.

[21] PHP: Hypertext Preprocessor. http://www.php.net/.

[22] Ruby on Rails. http://rubyonrails.org/.

[23] The Official Microsoft ASP.NET 2.0 Site. http://www.asp.net/.

[24] Transaction Processing Performance Council Benchmark W (TPC-W). http://www.tpc.org/tpcw/default.asp.