A Self-Adapting Web Server Architecture: Towards Higher Performance and Better Utilization

Farag Azzedin Khalid Al-Issa
Information and Computer Sciences Department
King Fahd University of Petroleum & Minerals
Dhahran, Saudi Arabia
fazzedin@kfupm.edu.sa khalid.issa@aramco.com

ABSTRACT

The way at which a Web server handles I/O operations has a significant impact on its performance. Servers that allow blocking for I/O operations are easier to implement, but exhibit less efficient utilization and limited scalability. On the other hand, servers that allow non-blocking I/O usually perform and scale better, but are not easy to implement and have limited functionality. This paper presents the design of a new, self-adapting Web server architecture that makes decisions on how future I/O operations would be handled based on load conditions. The results obtained from our implementation of this architecture indicate that it is capable of providing competitive performance and better utilization than comparable non-adaptive Web servers on different load levels.

KEYWORDS: Operating systems; Internet and Web computing; Synchronous and asynchronous I/O; Concurrency

1. INTRODUCTION

Performance is a vital factor behind the success of Web-based services. As a result, working towards improving the performance of Web servers becomes a critical issue. As a matter of fact, the tremendous growth of Web-based services and applications over the past several years, the growth of network bandwidth, and the presence of a very demanding, large and growing community of Web users, are expected to put more heat on Web servers [16] [5] [2] [19] [18].

In general, performance can be looked at as either *macro* or *micro* performance [20]. A Web server's macro performance refers to the side of performance observed by clients, including throughput and response time. Micro performance, on the other hand, represents the server's internal performance, including lots of metrics like clock Cycles Per Instruction (CPI) and cache hit rate. While both of the two classes of enhancement contribute to the overall performance of a Web server, they differ in complexity, significance and effectiveness. For instance, a simple approach towards improving the overall performance is to use replication, which is precisely used to give better macro performance by providing multiples of the original throughput. This approach, however, only provides a workaround that would still suffer from the same set of issues that exist in the server architecture [19]. A more effective alternative would be to look into enhancing a Web server's micro performance, which would not only improve the overall performance, but also allows for eliminating major limitations and defects. As an example, some earlier work has shown that a server would be able to satisfy its clients with better throughput and response time if its cache locality is improved [11], or if more clients requests are taken every time the server accepts new requests [5] [3].

The primary task of Web servers is to deliver Web contents in a concurrent fashion. In order to deliver contents, frequent disk I/O operations have to take place, and that hits concurrency significantly. Concurrency is generally achieved either through asynchronous system calls to avoid blocking the server, or through multiple server instances using threads or sub-processes in which case the use of synchronous system calls becomes acceptable. The former approach is typically used in the Single-Process Event-Driven (SPED) Web server architecture. As shown in Figure 1, a client request is first accepted by the SPED server process. Then, the server performs all necessary computations as requested by the client. In case I/O operations are needed, the server would enqueue these I/O requests against an asynchronous system call like *select()*, which should relieve the server from having to block waiting for the I/O operation to be completed. While the I/O is being performed, the server may start processing computations associated with another

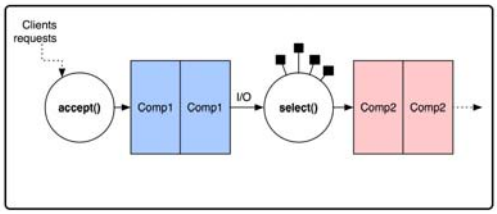

Figure 1. The SPED Web Server Architecture

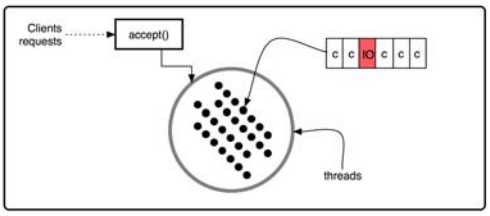

Figure 2. The Multi-Threaded Web Server Architecture

client request.

While the SPED model works pretty well when serving cached contents, it is not efficient when contents are to be fetched from disk, in which case the server is expected to interleave the serving of requests with slow I/O operations [16] [13]. In addition, asynchronous system calls available for use to implement concurrency in this model causes the server process to actually block in certain cases [16] [8]. The alternative for achieving concurrency through asynchronous calls is to run multiple server instances through either multiple processes or multiple threads. As shown in Figure 2, in a multi-threaded server, a thread is responsible for processing all computations as well as any needed I/O associated with the client request. While the multi-process and multi-threaded architectures are rather easier to implement [13] [4], they exhibit relatively low utilization of a server's resources since they allow processes and threads to block. In addition, the fact that every connection gets assigned a unique server thread or process has a negative impact on scalability. Moreover, the introduction of persistent connections in HTTP 1.1, which permits a connection to stay active while different objects are transferred, allowed for even less efficient utilization of the server [4].

The fact that both of the two alternatives have significant limitations in concurrently handling I/O operations and clients requests has motivated researchers to: (1) come up with hybrid architectures that combine certain features of the two approaches [16] [19] [4], (2) implement libraries to replace available, less efficient system calls [21] [8] [12], and (3) suggest and implement enhancements to these two architectures [11] [2] [5] [1] [9] [13] [3], . What is common about all the proposals we survey is that their technique for handling I/O operations is pre-determined, and can not adapt to the continuously changing state of the server. However, there are situations in which allowing a server thread to block is more cost-effective. There are also cases in which a server thread is so expensive that it should be allowed to only perform computations, and no I/O. As a result, we believe that a server should be allowed to determine what is best I/O scheme to follow based on load conditions.

This research work proposes the algorithm and structure of a self-adapting, multi-threaded server architecture that has the ability to switch between two different I/O schemes depending on load conditions. To the best of our knowledge, this is the first proposed adaptive Web server model that can follow more than one I/O scheme.

1.1. Motivation

Enhancing the way at which a Web server handles I/O has been an active topic in literature over the last decade. The key motivation was to propose ideas for higher concurrency, despite the long latency of disk I/O. Hybrid architectures, which rely on combinations of the two original approaches, are among the most interesting proposed ideas. For instance, one of the first hybrid models suggests employing a pool of helper processes whose role is to perform I/O. This relieves SPED servers from the need for inefficient asynchronous I/O, and greatly increases these servers' capacity.

While different hybrid models employed different techniques for achieving the primary goal of improving performance through increasing utilization and scalability, they are common in that they enforce a fixed I/O scenario. Proposed hybrid models that allowed blocking I/O to take place would allow that even under overloaded conditions. Similarly, models that utilize helper processes would pass I/O requests to helper processes even under lower load conditions, in which case blocking might both faster and less of an overhead. This motivates us to propose a self-adapting Web server architecture and evaluate its effectiveness to outperform non-adaptive architectures in both throughput and response time on different load conditions.

1.2. Objectives

The main objective of this research work is to outline two major limitations present in today's widely used I/O model in Web servers, the synchronous blocking I/O model. Scalability is highly affected due to allowing server threads to block for I/O. This would consequently have negative effects on a Web server's overall performance as it limits

throughput and increases response time. In addition, allowing server threads to perform blocking I/O operations represents an inefficient utilization of this valuable resource. Utilization becomes a critical issue as the rate of incoming clients requests increases while server threads are idle over I/O.

We still believe that blocking for I/O is the right choice under certain circumstances. Therefore, we propose in this paper a Web server architecture that can use both the blocking and the non-blocking I/O models under different load conditions to enhance performance and provide better utilization of server threads.

1.3. Contributions

This research work contributes to literature by first providing a survey and classification of current Web server architectures. Over the past several years, a number of architectures have been proposed to overcome limitations in the original models, as well as to improve performance and cope with the increasing popularity of Web-based services. In this research, we present a survey of these models along with a classification that is based on how they handle I/O requests.

Second, this research work brings to attention the need for adaptability in Web servers. This feature will enable the server to choose a more practical work scenario depending on past, current, or foreseeable circumstances.

Third, this research work promotes the use of asynchronous I/O for multi-threaded servers. The highly improved scalability obtained with this technique compared to the very common contender justifies it very well.

Last, this research work introduces a performance evaluation of an implementation of the proposed Web server architecture.

The paper is organized as follows: Section 2 presents a survey and a classification of existing Web server architectures. In Section 3, we explain the I/O models and outline strengths and weaknesses of each one of them. Section 4 describes the advantages of implementing self-adaptability in Web servers. Then, we describe the internals of the self-adapting Web server architecture in Section 5. In Section 6, we present the results obtained from our experiments in which we compare the performance of an implementation of the self-adapting Web server model to non-adaptive Web servers. We then explain how utilization is enhanced in the self-adapting Web server architecture in Section 7. Finally, Section 8 presents a conclusion of this paper, along with plans for future work.

Table 1. Some Existing Web Server Architectures

Year	Contribution	Class	Remarks
1999	AMPED	1	This is a SPED server that passes I/O requests to helper processes or threads
2001	Cohort scheduling	3	In this server, the order of executing threads is changed in order to execute similar computations consecutively, which reduces cache misses.
2001	SEDA	1	A pipelined server that consists of multiple stages, each is associated with a pool of threads.
2001	Multi-Accept	3	Instead of accepting a single incoming connection, a bulk of incoming connections are taken every time *accept() is called.*
2003	Cappriccio	2	A multi-threaded package that uses asynchronous I/O and provides high scalability.
2004	Lazy AIO	2	A new asynchronous I/O library that is meant to resolve issues with the available asynchronous libraries.
2005	Hybrid	1	A multi-threaded server that employs an event-dispatcher to resolve issues with allowing persistent HTTP connections.
2007	SYMPED	1	This server employs multiple SPED instances.
2008	MEANS	2	A software architecture that uses micro-threads for scheduling event-based tasks to Pthreads.

2. LITERATURE REVIEW

The multi-threaded and the event-based architectures are the original approaches for implementing a server. As each of the two has its own limitations and areas for improvement, many proposals over the last several years came to suggest and implement enhancements to overcome limitations and improve performance. We classify these proposals into three classes: (1) proposals for hybrid architectures, (2) proposals that suggest replacement libraries, and (3) proposals that disregard the I/O issue and focus on other aspects to improve performance. Table 1 summarizes our classification of the available Web server architectures.

2.1. Proposals for Hybrid Approaches

The first class of these proposals focused on deriving hybrid architectures that would combine features from the two original models. One of the early attempts was the asymmetric multi-process event-driven (AMPED) architecture [16], which provides a more effective solution for performing I/O operations in SPED servers, in which asynchronous system calls like *select()* are used. The AMPED

architecture is similar to SPED in that it typically runs a single thread of execution. However, instead of performing asynchronous I/O using descriptors through *select()*, AMPED passes I/O operations to helper processes. As a result, if blocking for I/O ever takes place, only helper processes will have to set idle and not the server's process.

A more recent hybrid model is the *Staged event-driven architecture* (SEDA) [19]. In SEDA, the entire work-flow of processing requests is re-structured into a sequence of stages, which makes it very similar to a simple pipeline. The main motivation behind introducing SEDA is to provide massive concurrency by allowing pools of threads to handle specific sets of tasks at the same time.

D. Carrera et. al. [4] proposes a solution for the case in which an idle client continues to hug a server thread, which is an undesirable consequence of allowing persistent connections in HTTP/1.1 for multi-threaded servers. To resolve this issue, they introduce a hybrid model in an event-dispatcher is used to identify sockets with readable contents and assign them to a server thread, which would read and process the request.

D. Pariag et. al [17] proposes the Symmetric Multi-Processor Event Driven (SYMPED) architecture. The SYMPED model consists of multiple SPED instances working together to increase the level of concurrency. Whenever one of these instances blocks for disk accesses, other instances can take over processing clients requests.

2.2. Proposals for Replacement Libraries

The second class of proposals started from the fact that available asynchronous system calls are not efficient, and suggested that they should be replaced. For instance, the way *select()* works requires it to block when used on disk I/O [16] [8]. Proposals in this area focus on providing replacements to these limited libraries. Elmeleegy et. al. [8] proposed Lazy Asynchronous I/O (LAIO), an asynchronous I/O interface to better support non-blocking I/O that would be more appropriate for event-driven programming. LAIO basically provides a non-blocking counterpart for each blocking system call. It handles blocking I/O operations for the application setting on top, while the application is allowed to move on with processing other requests.

Capriccio [1] is a scalable thread package that has the ability to scale up to 100,000 threads. This package was designed to resolve the scalability issue of the multi-threaded architecture. This high scalability in this solution was achieved through the use of *epoll()*, an asynchronous I/O interface that has proven to perform better than both the *select()* and the *poll()* interfaces with the right optimizations [9].

Lei et. al. [12], proposes MEANS, a micro-thread software architecture that consists of two thread-layers setting between the application and the operating system. An application that makes use of MEANS will assign work to MEANS micro-threads, which assign tasks in an event-based scenario to Pthreads interacting directly with the operating system.

2.3. Proposals for Modifying Other Architectural Components

The last class of proposals focused on enhancing the way the original models operate, while allowing the same I/O scenarios to take place. Chandra et. al. [5] and Brecht et. al. [3] suggest modifying the way a Web server accepts new connections. In the case of SPED, instead of accepting only a single new connection every time the server checks for incoming connections, they suggest accepting multiple connections [5]. This method increases the rate of accepting new connections, and increases concurrency of the server by providing more work that is ready to be processed at any instance.

Larus et. al. [11] suggest enhancing Web server performance by increasing locality and minimizing cache misses. They proposed a server model in which different requests are analyzed to identify similar computations. The order at which these requests are processed would be altered to allow similar computations to be processed consecutively. Executing similar computations as a group increases locality, and consequently improves performance.

3. I/O MODELS

Whenever an I/O operation needs to be performed by a Web server, the operating system actually takes care of it. This is due to many reasons, including maintaining a layer of security through which only privileged applications are granted access to certain files. While the I/O operation is being carried out, a server could either be blocked waiting for it to complete, or is free to process other requests. This depends entirely on the type of I/O the server initiated.

3.1. Synchronous I/O Operations

Synchronous I/O could be blocking or non-blocking to the calling process [10]. While both are performed through the same *read* and *write* system calls, the non-blocking requires the "O_NONBLOCK" option to be set when the *open()* system call is issued. The main issue with the synchronous non-blocking model is that it would require the calling process to send numerous calls to get the status of the requested I/O. As a result, this model is known to be extremely inefficient [10].

The synchronous blocking model, on the other hand,

is widely used in multi-threaded and multiprocess Web servers, where the presence of multiple instances of the server makes the undesirable blocking less significant. There are, however, major issues with this I/O model. Particularly for multi-threaded Web servers, allowing threads to block would largely limit the server scalability, leading to degraded performance with low throughput and high response time. In addition, allowing a server threads to block introduces idle time and makes inefficient utilization of valuable resources.

3.2. Asynchronous I/O Operations

Just like the case with synchronous I/O, asynchronous I/O can be either blocking or non-blocking [10]. A well-known example of the asynchronous blocking I/O scheme is the *select()* system call [7]. Through this system call, the calling process can add new asynchronous I/O requests and take the ones that are complete for processing. The *select()* system call keeps a list of file-descriptors for every request it is processing. Every time a *select()* is issued, the calling process is expected to block for a period of time to allow some I/O to complete.

In the asynchronous non-blocking system calls, the calling process returns immediately after initiating an I/O request. This class of I/O operations is performed through system calls present in the Asynchronous I/O (AIO) library, including the *aio_read()* and *aio_write()*. Once an AIO operation is initiated, it gets carried out to completion for the calling process, which would need to use some notification mechanisms to identify completed I/O requests. Two of the widely used notification mechanisms include signals and polling. In signals, the kernel would send a signal to the calling process once its requested I/O operation is complete. In polling, on the other hand, a dedicated thread is typically used to "poll" the status of a list of I/O requests passed to him by the calling process.

While in synchronous blocking I/O the calling process has a limit on how many requests it is processing at a given time, asynchronous I/O enables the calling process to accept much more work. This is largely due to the fact that a worker thread is relieved from having to handle a client requests to completion. This provides a boost to the server's scalability, and consequently enables the server to sustain higher throughput with lower response time. In addition, utilization of server threads is kept high since they are devoted for processing and not for sleeping over I/O. There are, however, issues with the asynchronous non-blocking approach. There are situations where blocking is more preferable. Under lower workloads, as well as for shorter I/O requests, blocking becomes a more convenient alternative and is expected to perform better. In addition, the need for notification mechanisms both complicates the rather simple I/O scenario and introduces an overhead that can have significant drawbacks. Last but not least, there are limits to how many AIO requests a kernel allows at a given time [6].

4. THE NEED FOR SELF-ADAPTABILITY

Existing Web servers follow a fixed I/O scheme at all times, regardless of the fact that different I/O operations are of varying cost, and that they are performed at different load conditions. For instance, if we look at a server that allows blocking I/O to take place, then we know that concurrent I/O requests would cause several threads to block until the I/O is complete. While this is acceptable under lower load conditions, it becomes undesirable as the server gets overloaded as that would highly limit the scalability of the server. The inefficient utilization is also an issue here since the server is allowed to set idle over I/O while other queue is building up.

The basic alternative of allowing only non-blocking I/O to take place has limitations too. While this may significantly enhance utilization and improve performance, it introduces the overhead of notification that might not be well justified under low load. For instance, polling is a notification technique in which an I/O request - initiated by a server's thread - is regularly checked for completion by a polling thread. When this server is under lower load conditions, then this I/O scenario becomes quite overwhelming. As a matter of fact, it would be more practical and reasonable to allow threads to block in such a case rather than introduce the overhead of asynchronous I/O.

This brings to attention the need for a Web server to adapt to different work scenarios under different load conditions in order to ensure best possible performance. More precisely, the use of blocking I/O is very practical whenever incoming requests are within a Web server's capacity. Beyond the server capacity, the use of non-blocking I/O increases scalability, giving better chances for higher performance, and utilizes server threads highly and more appropriately.

5. THE SELF-ADAPTING WEB SERVER ARCHITECTURE

The self-adapting Web server architecture is similar to the multi-threaded Web server model, with the distinction that it allows for two types of I/O: synchronous blocking, and asynchronous non-blocking I/O operations. Synchronous I/O takes less effort from the server itself, but lead to limiting both utilization and scalability. Asynchronous I/O comes with its own overhead, but is very useful at times when we need the best scalability and utilization of server

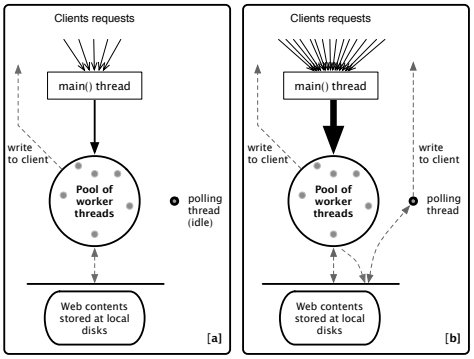

Figure 3. The Self-Adapting Web Server Work Scenario

resources. The criteria of adaptability can follow more than one pattern, as long as it includes well defined and easily detectable conditions. For example, in our implementation we use the presence of queued requests as an indicator of overload.

In our implementation of the self-adapting server model, we used the following pools of threads explained in [15]: a pool of 64 worker threads, and a pool of a single polling thread for detecting completed asynchronous I/O requests. While the use of asynchronous I/O is not very common in multi-threaded servers, it has many advantages. With synchronous I/O, the maximum number of requests to be serviced at anytime would equal the number of threads. However, using asynchronous I/O, worker threads are able to initiate I/O requests and move on to servicing other client requests, and that would greatly improve scalability. The default I/O scheme for our server is the synchronous blocking, and the server would only switch to using asynchronous I/O as load increases (see Figure 3).

As can be seen in Figure 3 client requests are expected to get assigned to worker threads almost instantly under lower load conditions, without having to wait for long periods of time in the queue. As the rate of incoming requests increases, less worker threads become free and we get to a point at which all worker threads are busy processing client requests. At this point, incoming requests start piling up in the queue of the thread pool, and the server would notice that and instruct worker threads to rather perform the next I/O requests as asynchronous non-blocking operations. This will continue to be the case until all queued requests are assigned to worker threads, and then the server may resume the default working scenario of using synchronous I/O scheme.

Figure 4 shows a flowchart of the algorithm used to build our server. The upper-most block in the diagram, labeled (a),

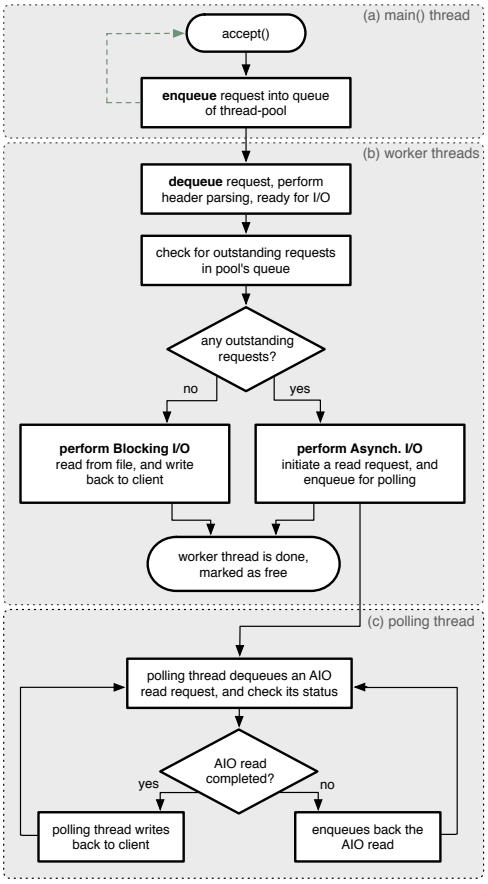

Figure 4. Adaptability Algorithm in the Self-Adapting Web Server Architecture

represents the work done by the main thread. This thread is basically the server's dispatcher, which binds to the listening port and continuously assign incoming requests to the queue of the threads-pool. The block in the middle of the diagram shows the work done by a worker thread. First, the worker thread dequeues a work item from the queue, and starts processing it. When it gets to the point where it needs to perform I/O, the worker thread will have to check for the presence of outstanding requests in the queue, and if there are none, then it would perform a blocking I/O operation. In case the queue is not empty, however, then that is a good indicator that all other worker threads are busy and that the server is about to be - or already is - overloaded. As a result, the worker thread would initiate an asynchronous I/O request, and would enqueue a work item associated with it for the polling thread. That moves us to the third block in the diagram, labeled (c), which represents the work done by the polling thread. Once the polling thread dequeues this work item, it would be able to identify the I/O request in

charge, and would be able to check on its status. In case the I/O is found to be complete, then the polling thread will respond back to the client, otherwise it enqueues back for later.

5.1. Implementation

The self-adapting Web server architecture employs both synchronous and asynchronous system calls for making I/O. For synchronous I/O, it uses the system's *read()* and *write()*. For asynchronous I/O, it uses the Asynchronous I/O (AIO) set of library calls, which is included in the 2.6 Linux kernel. The main reason we elected to use this library is that it includes true non-blocking calls like the *aio_read()*, which our worker threads use to initiate an asynchronous I/O before it moves on to servicing another request.

The programming language in use for implementing the server was C, and the code was compiled using GCC.

6. PERFORMANCE EVALUATION

In order to evaluate the performance of the self-adapting Web server model, we compare it to two non-adaptive servers we developed. The three Web servers we developed differ only in the way I/O is performed, while everything else is kept the same. In addition, we compare our model to a similarly configured version of the Apache server, which represents the state-of-the art in Web servers. The goal of comparing the servers' performance to Apache is to provide a sense of how it performs compared to a fully optimized, production-ready server such as Apache, under the same workload.

6.1. Performance Metrics

Web servers performance is usually measured in terms of both throughput and response time. Throughput of a Web server is given by:

$$Throughput = \frac{R}{T} \quad (1)$$

where **R** represents the number of successfully completed requests, and **T** represents the total time that was needed to complete them. Throughput provides a measure of how many requests a Web server can successfully complete in a unit of time. The other metrics, response time, represents the time between the completion of a request and the beginning of a response, and is often measured in milli-seconds.

6.2. Testing Environment

Besides the self-adapting Web server, which we explained in Section 5, we implemented a multi-threaded server that is blocking I/O (BIO)-based. This server is composed of a main thread, and a pool of worker threads. The main thread binds itself to the listening port and then works as a dispatcher assigning incoming clients requests to worker threads, which are responsible for processing all requests to completion. This processing includes parsing the HTTP header, validating the request, performing blocking I/O requests to obtain the requested Web files, and sending the responses back to clients.

We also implemented a multi-threaded server that is asynchronous I/O (AIO)-based. In addition to the main thread and the pool of worker threads available in the BIO-based server, the AIO-based server has a dedicated polling-thread whose role is to verify completion of initiated I/O requests. In this server, the main thread works similar to the one of the BIO-based server, but the way worker threads work differs. The rule of thumb here is that no worker thread is allowed to perform I/O operations. As a result, when an I/O operation is needed, a worker thread only initiates it and then is freed up for processing another client request. Of course, somebody else will have to follow up with the I/O operation that has been just initiated, and this is the job of the polling thread. More precisely, worker threads use the aio_read() call to initiate an I/O read operation. Since this call is non-blocking, the thread is expected to return from it immediately. Just before it is allowed to to go back to the pool, the worker thread enqueues a request for the polling-thread to track the initiated I/O operation. After that, the polling-thread will dequeue the request and check the status of its associated I/O operation. If the I/O is complete, the polling-thread will respond back to the client. Otherwise, it would enqueue the request back to be re-checked later.

Apache is a well-known Web server that follows the original multi-threaded model, with blocking I/O operations performed as needed. In this experiment, Apache was reconfigured to match the setup of the other three servers in terms of the number of processes and threads. This provides a more reasonable and fair comparison between the models included in the experiment. However, we have to emphasize that Apache is rather an optimized server that includes many features, while the other three servers are rather simple.

The benchmarking was carried out in the EXPEC Computer Center (ECC) in Saudi ARAMCO. The hardware used for benchmarking included 11 rack-mounted cluster nodes forming a server and 10 clients. Every node had dual 2.6 GHz AMD Optron processors, 8 GB of memory, and gigabit Ethernet interfaces. Workloads were generated using Httperf [14], a well-known and effective synthetic workload generator that can make HTTP requests as fast as a Web server can handle them.

Table 2. Results of the First Experiment

Model	Best throughput	corresponding response time
BIO	1378.46	2.7
AIO	1645.89	47.8
Self-adapting	1987.4	74.1
Apache	1999.7	16.4

Two sets of experiments were conducted. In the first set, the 10 clients repeatedly issue twenty thousands requests for a single 10 KB HTML file at rates ranging from 100 requests per second to 2000 requests per second. This part of the experiment allows the servers to perform at their highest capacity since the file will most likely going to be available in cache. The second set of experiments is similar, but the content of the 10 KB file is continuously altered to enforce disk I/O and to evaluate effects of that on the overall performance.

6.3. Results

In the first experiment, the 10 KB HTML file is expected to be repeatedly fetched from cache since its contents are kept the same. Results from the first experiment (shown in Figure 5 and Figure 6) indicate that Apache continues to work impressively well under higher workload, both in terms of throughput and response time. Apache was able to provide throughput of as high as approximately 2000 replies per second, with response time that never exceeded 20 milliseconds. The blocking I/O-based server performed less efficiently than the rest of the servers included in our testing, and was able to provide throughput as high as approximately 1379 replies per second. The fall in throughput after this point was accompanied by a significantly high increase in response time of up to more than 300 milliseconds. The non-blocking AIO-based server performed better, and was able to provide throughput as high as approximately 1645 replies per second. Response time in the AIO-based server was within 60 milliseconds even under higher workloads. Finally, our self-adapting server performed much better than both the BIO and AIO-based servers in this experiment, and performed very similar to Apache in terms of throughput. It was able to provide throughput as high as about 1987 replies per second, but response time was slightly higher than the AIO-based server under higher workloads. The higher response time under higher workload conditions is primarily caused by the presence of the less efficient BIO-based portion. Table 2 summarizes the results obtained from this experiment.

In the second experiment, the 10 KB HTML file is loaded from disk every time it is asked for since its content is

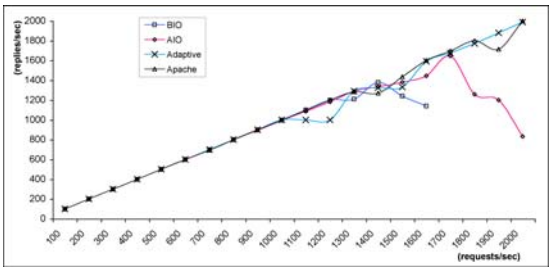

Figure 5. Throughput in the First Experiment

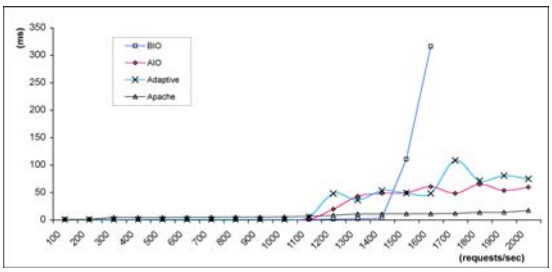

Figure 6. Response Time in the First Experiment

continuously altered at a high rate. Results obtained from this experiment are shown in Figure 7 and Figure 8. As can be seen from these figures, Apache was also able to perform best in this part, with best throughput of approximately 1856 replies per second, and response times less than 15 milliseconds in all cases. The blocking I/O-based server performed least in this experiment as well, with best throughput of approximately 1295 replies per second. The server's response time beyond this point was very high, similar to what has been seen in the first experiment. The AIO-based server performed better with best throughput of approximately 1498 replies/second, and response time within 60 milliseconds. Finally, the self-adapting server also performed better than both the BIO- and AIO-based in this experiment, with throughput of as high as approximately 1573 replies/second and highest response time of approximately 51.8 milliseconds. The presence of the blocking I/O portion in the self-adapting server has a positive impact in keeping response time low under lower load conditions. However, that caused response time to go higher under higher workloads, but without jeopardizing the higher throughput. Table 3 summarizes the results obtained from this experiment.

It can be seen from the two experiments that the self-adapting server was able to take advantage of the presence of the two different I/O schemes to deliver more at the time a non-adapting server would saturate.

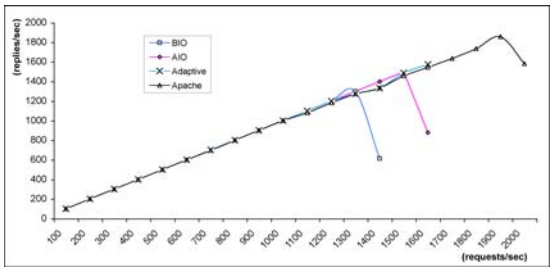

Figure 7. Throughput in the Second Experiment

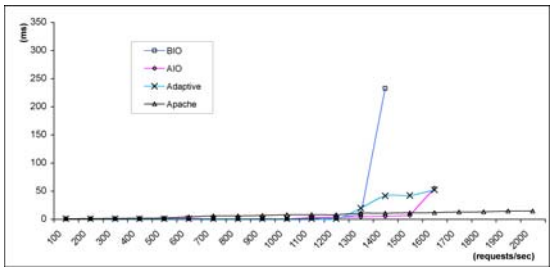

Figure 8. Response Time in the Second Experiment

7. UTILIZATION

In the blocking I/O model, a server is expected to stay idle while I/O is being performed for it. This represents a major drawback of this model, as the server is expected to be busy processing clients requests. The issue becomes more significant if the server is already under higher load conditions, where there would be many requests waiting to be processed by a server that is idle waiting for I/O.

In the asynchronous non-blocking I/O model, on the other hand, a worker thread is completely devoted for processing at all times. While this approach makes much better utilization of server threads, it imposes some limitations under certain conditions. For instance, enforcing this scenario under lower workload conditions - where not all worker threads are busy - might not be practical. As a matter of fact, considering the overhead of asynchronous notification, it might be more convenient to allow a worker thread to block for I/O rather than passing it to a polling thread to continuously check for its status among tens of other I/O requests.

The utilization of worker threads in the self-adapting Web server architecture is highly moderate. Whenever blocking for I/O starts becoming significant to the overall performance, the server switches to performing non-blocking I/O operations. More precisely, the server allows threads to block for I/O as long as there are no requests waiting to be processed. Once requests start getting delayed in the queue due to unavailability of worker threads, utilization of these threads becomes more significant. To increase utilization in this case, the server will elect to perform future I/O requests as asynchronous non-blocking I/O.

Table 3. Results of the Second Experiment

Model	Best throughput	corresponding response time
BIO	1295.1	8.3
AIO	1498.7	6.3
Self-adapting	1573.6	51.8
Apache	1856.4	14.1

8. CONCLUSION AND FUTURE WORK

In this paper, we presented a new, self-adapting Web server architecture that can switch between two different I/O models based on workload conditions. Under higher workload conditions, asynchronous non-blocking I/O increases the server scalability and thus performance, and makes better utilization of its resources. Under low workloads, however, blocking I/O is both more convenient as well as more efficient. In our implementation of this architecture, we designed a server that detects outstanding requests piling up in its queue and would accordingly switch from performing blocking I/O to non-blocking I/O for future requests. Results obtained show that this model pushes the limit of our simple servers and was able to get closer in performance to a similarly configured Apache Web server.

While adapting to load conditions makes sense, as it provides justifiable switching point between the two I/O schemes, there are more logical adaptation conditions. For instance, disk I/O operations are impacted by how busy the disk is. As a result, a server could switch to performing asynchronous I/O whenever the disk performance degrades, in order to keep more free worker threads. As future work, more adaptation conditions would need to be implemented. In addition, adaptability would need to be implemented on a production-ready Web server like Apache.

ACKNOWLEDGEMENTS

The authors would like to thank the EXPEC Computer Center (ECC) in Saudi ARAMCO for providing them access to ECC computer clusters for the benchmarks in this paper. In addition, they would like to thank all anonymous reviewers for their helpful comments, resulting in a significant improvement in the quality of this manuscript. This research is supported by King Fahd University of Petroleum and Minerals under Research Grants IN080430 and FT060028.

REFERENCES

[1] R. V. Behren and J. Condit and F. Zhou and G. C. Necula and E. Brewer, “Capriccio: Scalable Threads for Internet Services,” Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP ’03), Oct. 2003, pp. 268-281.

[2] T. Brecht and M. Ostrowski, “Exploring The Performance of Select-Based Internet Servers,” Technical Report HPL-2001-314, HP Labs, Nov. 2001.

[3] T. Brecht and D. Pariag and L. Gammo, “Accept()able Strategies for Improving Web Server Performance,” Proceedings of USENIX 2004 Annual Technical Conference (USENIX ’04), Jun. 2004.

[4] D. Carrera and V. Beltran and J. Torres and E. Ayguade, “A Hybrid Web Server Architecture for e-Commerce Applications,” the 11th International Conference on Parallel and Distributed Systems (ICPADS’05), 2005, pp. 182-188.

[5] A. Chandra and D. Mosberger, “Scalability of Linux Event-Dispatch Mechanisms,” Proceedings of USENIX 2001 Annual Technical Conference (USENIX ’01), 2001, pp. 231244.

[6] D. Chisnall. POSIX Asynchronous I/O. Available: http://www.informit.com/

[7] D. E. Comer and D.L. Stevens, INTERNETWORKING WITH TCP/IP VOL III: CLIENT-SERVER PROGRAMMING AND APPLICATIONS, Prentice Hall”, Englewood Cliffs, New Jersey, 2000.

[8] K. Elmeleegy and A. Chanda and A. L. Cox and W. Zwaenepoel, “Lazy Asynchronous I/O for Event-Driven Servers,” Proceedings of USENIX 2004 Annual Technical Conference (USENIX ’04), 2004, pp. 241-254

[9] L. Gammo and T. Brecht and A. Shukla and D. Pariag, “Comparing and Evaluating epoll, select, and poll Event Mechanisms,” the 6th Annual Ottawa Linux Symposium, Jul. 2004.

[10] M. T. Jones. Boost application performance using asynchronous I/O. Available: http://www.ibm.com/developerworks/linux/library/l-async/

[11] J. Larus and M. Parkes, “Using Cohort Scheduling to Enhance Server Performance,” Proceedings of USENIX 2002 Annual Technical Conference (USENIX ’02), Jun. 2002, pp. 103-114.

[12] Y. Lei and W. Zhang and Y. Gong and H. Zhang, “MEANS: A Micro-Thread Architecture for Network Server,” the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), Feb. 2008, pp. 38-42.

[13] H. R. Liu and T. F. Chen, “Scalable Locality-Aware Event Dispatching Mechanism for Network Servers,” the IEE Proceedings - Software, Vol. 151, No. 3, Jun. 2004, pp. 129-137.

[14] D. Mosberger and T. Jin, “Httperf: A Tool for Measuring Web Server Performance,” the First Workshop on Internet Server Performance, Jun. 1998.

[15] B. Nichols and D. Buttlar and J. P. Farrell, PTHREADS PROGRAMMING: A POSIX STANDARD FOR BETTER MULTIPROCESSING, O’Reilly & Associates, Inc, 1996.

[16] V. S. Pai and P. Druschel and W. Zwaenepoel, “Flash: An Efficient and Portable Web Server,” Proceedings of USENIX 1999 Annual Technical Conference, Jun. 1999, pp. 199-212.

[17] D. Pariag and T. Brecht and A. Harji and P. Buhr and A. Shukla, “Comparing the Performance of Web Server Architectures,” the 2007 EuroSys Conference, Mar. 2007.

[18] K. Po and I. Sin, “Evaluation of Web Server Architectures,” Department of Electrical and Computer Engineering, University of Toronto, 2005.

[19] M. Welsh and D. Culler and E. Brewer, “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services,” Proceedings of the 18th Symposium on Operating Systems Principles (SOSP 2001), Oct. 2001.

[20] H. Xie and L. Bhuyan and Y. K. Chang, “Benchmarking Web Server Architectures: A Simulation Study on Micro Performance,” Proceedings of the 5th Workshop on Computer Architecture Evaluation using Commercial Workloads, Feb. 2002.

[21] N. Zeldovich and A. Yip and F. Dabek and R. T. Morris and D. Mazieres and F. Kaashoek, “Multiprocessor Support for Event-Driven Programs,” Proceedings of USENIX 2003 Annual Technical Conference (USENIX ’03), Jun. 2003.