Reducing Data Copying Overhead in Web Servers

by

Gary Yeung

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2010

Abstract

Web servers that generate dynamic content are widely used in the development of Internet applications. With the Internet highly connected to people's lifestyles, the service requirements of Internet applications have increased significantly. This increasing trend intensifies the need to improve server performance in dynamic content generation.

In this thesis, we describe the opportunity to improve server performance by co-locating the web server and the application server on the same machine. We identify related work and discuss their respective advantages and deficiencies. We then introduce and explain our technique that passes the client socket's file descriptor from the web server process to the application server. This allows the application server to reply to the client directly, reducing the amount of data copied and improving server performance. Experiments were designed to evaluate the performance of this technique and provide a detailed analysis of processor time and data copying during response delivery. A performance comparison against alternative approaches has been performed. We analyze the results to understand factors in data copying efficiency and determine that cache misses are an important factor in server performance.

There are four major contributions in this thesis. First, we show that in multiprocessor environments, co-locating web servers and application servers can take advantage of faster communication. Second, we introduce a new technique that reduces the amount of data copied by two-thirds. This technique requires no modifications to the application server code (other existing techniques do), and it is also applicable in a variety of systems, allowing easy adoption in production environments. Third, we provide a performance comparison against other approaches and raise questions regarding data copying efficiency. Our technique attains an average peak throughput of 1.27 times the FastCGI with Unix domain sockets in both uniprocessor and multiprocessor environments. Finally, our analysis on the effect of cache misses on server performance provides valuable insights into why these benefits are obtained.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Tim Brecht for his guidance, patience, understanding and support. This work would not be possible without his valuable insights and rational questioning in our discussion sessions. I am grateful for his patience and understanding during the difficult times. I would also like to thank Professor Peter Buhr and Professor Martin Karsten for their time and invaluable comments on my thesis work.

I am grateful for the scholarship provided by the National Science and Engineering Research Council and the financial support provided by Professor Tim Brecht and University of Waterloo.

Also, I would like to thank my good friends Betsy McIntyre, Michael Kwok, Philip Lai, Brian Lin, Jeffery Lui, James She, Kenneth Tsui, William Wong and Wendy Yuen who supported me through the good times and the challenging times.

My deepest gratitude goes to Jane for her love, understanding and support. Her unbiased comments helped me understand my weaknesses, her positive attitude encouraged me to face adversity and her smiles made all the difficult times bearable.

Last but not least, I would like to dedicate this thesis to my parents, Sam and Ada Yeung for their unconditional love, support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

1.1 Thesis Statement

This thesis explores techniques to improve the performance of web servers that are serving dynamic content. In particular, we evaluate a technique that can be used when the web server and the application server are co-located on the same machine (e.g., in a multiprocessor or multi-core environment). Our technique passes the client file descriptor from the web server to the application server permitting the application server to reply directly to the client. As a result, data does not have to be sent through the web server.

1.2 Motivation

Internet technologies have advanced rapidly in the last decade. These technologies have enabled us to incorporate the Internet into our daily lives. As the accessibility of the Internet increases, Internet content has also grown larger and become more complex. As a result, server performance has become an important area of study as it directly affects the cost of content delivery. Many companies currently use replication and load-balancing techniques to handle the increasing performance demand.

While there has been a fair amount of research on static content delivery, until recently, very few studies have examined the performance of dynamic content generation architectures. Also, very few have examined the performance of dynamic content generation in multiprocessor environments. Suzumura *et al.* [39] recently studied this problem in a uniprocessor setting and noticed a significant amount of processor time is spent on data copying.

Web servers and application servers were originally designed to be located on different machines and to communicate through Internet sockets. With advances in multiprocessor and multi-core architectures, we believe there are performance advantages to be gained by co-locating the web server and the application server on the same machine. Additionally, this co-location provides an opportunity to further reduce communication overhead with Unix domain sockets and shared memory. In this thesis, we propose a technique that passes the client socket descriptor from the web server to the application server. This technique allows the application server to bypass the web server by sending the response directly to the client. This approach in turn reduces the amount of data copied during response delivery, and improves server throughput. We then evaluate the performance of dynamic content generation architectures in both uniprocessor and multiprocessor environments. Finally, we conduct a detailed analysis of data copying operations to understand factors affecting server performance.

1.3 Contributions

Our work explains the advantages of co-locating the web servers and the application servers on the same machine in multiprocessor environments, which provides opportunities to improve web server performance. We present a technique to improve the performance of web servers that are serving dynamic content. We compare the performance of our technique against a technique that shares memory between the web server and the application server, and also against the original socket communication architecture in both uniprocessor and microprocessor environments. In our analysis, we carefully examine copying costs and processor cache statistics to gain a better understanding of the factors affecting server performance.

This thesis makes the following contributions:

- We show that in multiprocessor environments, co-locating web servers and application servers can take advantage of faster communication.
- We present a new technique that passes the client socket's file descriptor from the web server process to the application server. This technique allows the application server to send replies directly to the client, thereby bypassing the

web server, reducing the amount of data copied and decreasing the load on the web server. Our solution requires only minor modifications to the web server and FastCGI library. No changes to the application server or application server code are required. This solution can be applied in different operating systems, web servers, and application servers. When compared to the original FastCGI architecture, this technique reduces the amount of data copied by two-thirds.

- Our experiments in both uniprocessor and multiprocessor environments show that our technique and the approach that shares memory between the web server and the application server attain an average peak throughput of 1.27 times and 1.14 times the average peak throughput of the original architecture, respectively.

- In comparing our approach of passing the client socket's file descriptor from the web server to the application server against the approach of sharing memory between the web server and the application server, we notice that these techniques reduce the same amount of data copied, but our approach produce better performance. With further analysis, we determine that processor cache pressure is a major factor affecting data copying efficiency.

1.4 Outline

Chapter 2 provides background related to dynamic content generation and introduces the FastCGI architecture. Then, it discusses the benefits of co-locating the web server and the application server on the same machine in a multiprocessor environment and describes related work. Chapter 3 describes the opportunity to reduce the amount of data copied during response delivery in the FastCGI architecture. We present our solution that passes the client socket descriptor from the web server to the application server allowing the application server to reply directly to the client. Then, we explain how our technique is integrated within the FastCGI architecture and discuss the advantages and implementation details of our technique. Finally, we describe how our solution can be applied in different operating systems, web servers and application servers. Chapter 4 describes the experimental goals and setup, server architectures and workload used for performance evaluation. Chapter 5 compares the performance of both our technique and an approach that shares memory between the web server and the application server against the performance of the original socket communicating FastCGI architecture. We then further investigate data copying costs in uniprocessor and multiprocessor environments using processor cache statistics to answer several questions that arise in our performance analysis. Chapter 6 summarizes our work and discusses directions for future work.

Chapter 2

Background and Related Work

Web applications have grown rapidly since the days of static HTML. In the early days, the World Wide Web (Web) was an interlinked collection of documents and images. It started as a platform to share data and information with uni-directional information retrieval being the main usage pattern. Soon, the development of the e-commerce industry introduced interaction into the Web. This increased level of interaction has driven the need for dynamic content generation. Over the last decade, the growing use of the Web has driven a multitude of performance studies aimed at improving the response time of dynamic content generation.

2.1 Generating Dynamic Responses

To understand the early dynamic response generation architectures, we need to understand the functions of web servers. Web servers were initially designed to handle the retrieval of static documents. The generation of dynamic responses is typically not built into the web server but instead developed as a separate process or application, called an application server. Upon receiving a client's request (Figure 2.1, Step 1), the web server forwards the request to an application server (Step 2). The application server then performs computations and dynamically generates a response (Step 3), which is sent back to the web server (Step 4). Finally, the web server sends it to the client (Step 5). The request size is usually much smaller than the response size. We emphasize this visually in Figure 2.1. Steps 1 and 2 show a small request size and Steps 4 and 5 show a large response size.

Since the application server is separate from the web server, they need to exchange information through a communication mechanism such as, the Common

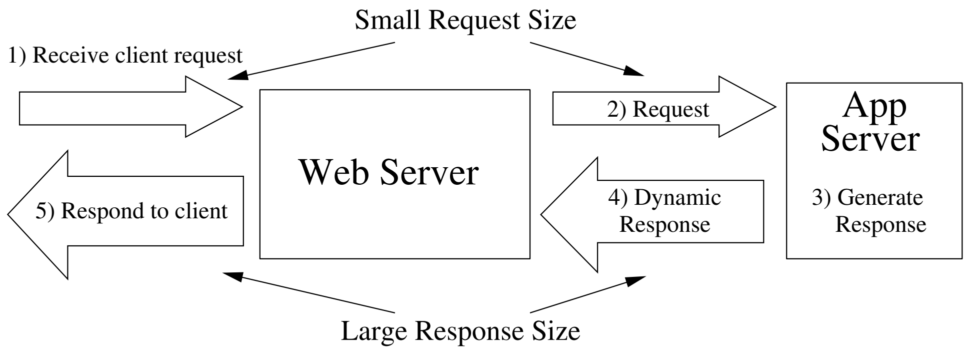


Figure 2.1: Dynamic Response Architecture

Gateway Interface (CGI).

2.1.1 The Common Gateway Interface

The Common Gateway Interface (CGI) [33] is a standard interface for information servers to execute external applications in real-time. With CGI, web servers can start an application server (as a separate process) in real-time and on demand to compute a dynamic response (Figure 2.2).

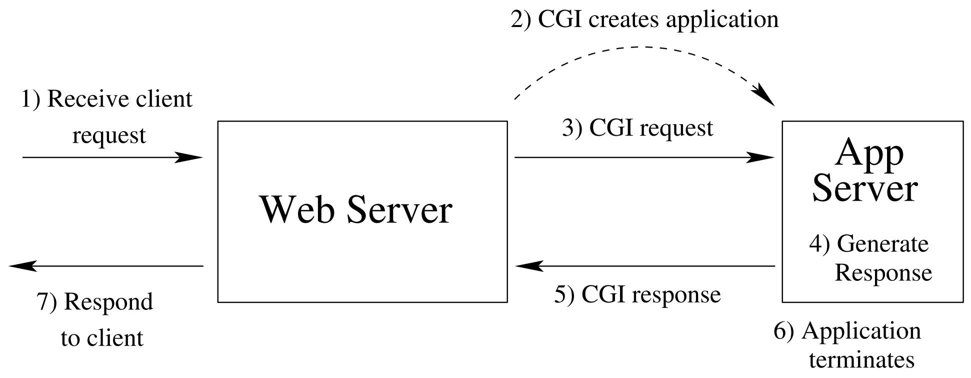


Figure 2.2: Common Gateway Interface

CGI provides a well-defined interface for dynamic web application development. It achieves language independence by defining a language-independent communication protocol; thereby, allowing applications to be developed in different programming languages. Process isolation in CGI eliminates the chance of buggy applications affecting the operation of the web server.

However, there are significant drawbacks to CGI. Since the application is developed as a separate process, each dynamic request requires the web server to ask the operating system to create an application server process (Figure 2.2, Step 2).

After the application server has completed sending its response to the web server, the application server terminates and the operating system reclaims its resources (Figure 2.2, Step 6). The operating system overhead in application server creation and termination is often significant compared to the actual computation by the application server [26]. The FastCGI architecture was proposed to eliminate these overheads [26, 12].

2.1.2 FastCGI

FastCGI [26, 12] was designed to address performance problems in CGI while retaining its benefits. It is supported by commonly available web servers such as recent versions of the IBM HTTP Server [14], Apache HTTP Server [9], Microsoft Internet Information Services (IIS) [43] and the Sun Java System Web Server [44]. It is also supported by many application servers including the IBM WebSphere Application Server [11], ESXX [23] and other application servers written in popular scripting languages such as Perl, PHP and Python.

To address the operating system overhead in CGI, rather than creating one application server per client request, the web server creates a pool of persistent FastCGI application servers at start up (Figure 2.3, Step 0). These application servers utilize a FastCGI library to communicate with the web server over a socket using the FastCGI protocol (Figure 2.3, Steps 2 and 4). Upon completion of a request, the FastCGI application server waits for a new request from the web server rather than terminating. FastCGI improves server performance by eliminating the operating system overheads of application server creation and termination.

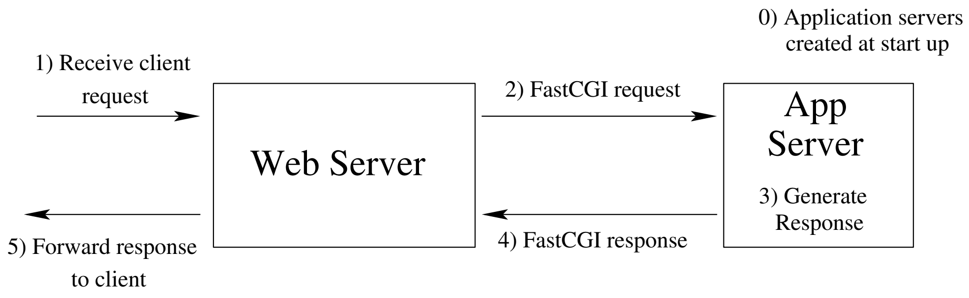

0) Application servers created at start up
1) Receive client request
2) FastCGI request
App Server
Web Server
3) Generate Response
4) FastCGI response
5) Forward response to client

Figure 2.3: FastCGI

FastCGI uses a socket to facilitate communication between the web server and the application server. Using a socket provides the flexibility to run applications on a separate machine from the web server in order to distribute load [26]. However, with increases in processor performance, availability of multi-core processors and large amounts of memory in current server machines, it is possible to co-locate the web server and the application server on the same machine. Server machines can now process data at very high rates, so distributing application servers requires a great investment in network switches and network cards to maintain good performance. Co-locating the web server and the application servers on the same machine is economical and provides good performance. Inter-process communication is much faster than network communication since it does not experience any network delay. Additionally, co-location allows for better utilization of processor resources. As described above, the web server is responsible for handling client requests and the application server is responsible for content generation. So the loads on these two components are seldom equal and CPU resources are often not fully utilized on one of the two systems. Putting them on the same machine allows processor resources to be better load-balanced. Therefore, we believe it is beneficial to host the FastCGI architecture on the same machine to take advantage of faster communication and improve server utilization.

Nonetheless, overhead is incurred when the web server and the application server are hosted on a single system. Since the web server and the application server are separate processes, one notable overhead is redundant data copying during inter-process communications. We further explain redundant data copying in Section 3.1.1.

In the following section, we describe related work that improves the performance of the communication between the web server and the application server within the context of the FastCGI architecture. In Chapter 3, we focus on the problem of data copying redundancy during response delivery. First, we formulate the problem in detail. Then, we describe our solution to reduce data copying and explain the improvements and benefits of our method. For the remainder of the thesis, our further discussion of the FastCGI architecture, related work and our proposed solution assumes the web servers and the application servers are co-located on the same machine.

2.2 Related Work

In the previous section, we covered the general architectures of dynamic content generation. We also noted some of the benefits of, and problems with, these architectures. In this section, we discuss work related to improving the performance of communication between the web server and the application server within the context of the FastCGI architecture.

2.2.1 User Space vs Kernel Space Servers

The high performance TUX [32] kernel space web server was introduced to improve the performance of static content delivery by avoiding overhead in context-switching and event notification. When the web server is implemented as a user process, each response is copied between user and kernel space at least once. TUX is a kernel space web server that eliminates such copying, thereby improving performance by reducing redundant data copying [19].

With the increasing demand of dynamic content generation, Shukla *et al.* [36] compared the performance of a kernel space web server to the performance of a user space web server in the context of dynamic content generation. Their results showed that as the proportion of dynamic requests increases, the performance gap between user space and kernel space servers diminishes. The authors believe dynamic content generation is processor-intensive; therefore, optimizations in TUX do not provide significant performance increases when compared to the performance increases obtained for static content delivery.

Given the lack of performance improvement for dynamic content delivery in TUX, its security and availability problems overshadow its benefits. Kernel space web servers run with unlimited privileges. Therefore, if the web server is compromised, the machine can be taken control of. Secondly, one can quickly restart a user space web server process when it crashes without affecting other web server processes on the same machine. But kernel space server crashes render the machine unusable and require a machine restart, thereby decreasing availability.

2.2.2 Data Copying Redundancy in Dynamic Content

Suzumura *et al.* [39] recently studied redundant data copying in their FastCGI implementation. Under the SPECweb2005 [5] e-commerce and support workloads, they reported that data copying constitutes 15.7% and 12.4% of processor time, respectively. Since dynamic content generation is often processor-bound, reducing the processor demand due to data copying should benefit overall performance.

They propose an optimization to reduce data copying, which can be described in three parts. First, they introduce a new type of character string object in PHP, called a file-type character string (FTCS) object to encapsulate the contents of a file. Second, the optimization makes use of a special header in the FastCGI response to pass the FTCS objects to the web server. Finally, the web server extracts the FTCS objects from the FastCGI response and sends the content using *sendfile()* [41, 27] with HTTP chunked transfer encoding.

Unlike the usual character string objects, which hold the entire contents of the file, an FTCS object holds only the file name. They modify the *file_get_contents()* function in the PHP runtime library to return an FTCS object instead of a string object with the file content. Figure 2.4 shows an example of PHP code in the application server. Two FTCS objects are returned by the *file_get_contents()* function in Lines 3 and 5. These FTCS objects are handled specially in the FastCGI response message to reduce data copying.

```
1: <?php
2: echo hello;
3: echo file_get_contents(/tmp/A.htm); // FTCS
4: echo world;
5: echo file_get_contents(/tmp/B.htm); // FTCS
6: echo bye;
7: ?>
```

Figure 2.4: Suzumura [39] Zero-Copy Code Example

1: **X-ZeroCopy:5/10:21/10**

2: hello**/tmp/A.htm**world**/tmp/B.htm**bye

Figure 2.5: Suzumura [39] Zero-Copy FastCGI Response Example

To encapsulate the FTCS objects in the FastCGI response, they introduce a new header in the FastCGI response protocol. In the FastCGI response message to the web server, the file name in an FTCS object is put in the message instead of the file content, thus reducing the amount of data sent between the application server and the web server. Figure 2.5 shows the FastCGI response generated by the example code in Figure 2.4. A new X-ZeroCopy header (Figure 2.5, Line 1) is included in the message to specify the locations of all the FTCS objects within the FastCGI response message.

To demonstrate the encapsulation of FTCS objects in the FastCGI response, we show the corresponding FastCGI response message for Figure 2.4, Lines 2 and

3. Output of any non-FTCS object (Figure 2.4, Line 2) remains the same, so the string content of the object is appended to the FastCGI response message (first 5 characters in Figure 2.5, Line 2). However, output of an FTCS object (Figure 2.4, Line 3) places its file name in the FastCGI response message and updates the X-ZeroCopy header with the location in the byte stream and length of the file name. For example, the file name "/tmp/A.htm" is appended to the FastCGI response message in Line 2 of Figure 2.5. Then, "5/10" is appended in Line 1 of Figure 2.5, indicating that the file name begins at index 5 of the FastCGI response message (Figure 2.5, Line 2) with a length of 10 characters. Since the file name is usually much shorter than the file contents, the size of the FastCGI response message is much smaller. Therefore, the amount of data being copied is greatly reduced which in turn reduces the processor usage required to perform the data copying.

After receiving the FastCGI response message, the web server parses the X-ZeroCopy header information to extract the file names of all FTCS objects. It then sends the response in chunks using HTTP chunked transfer encoding. Figure 2.6 shows the chunks from this example and the method used to send each chunk of the content.

Chunk	Method	Parameter
1	send	hello
2	sendfile	/tmp/A.htm
3	send	world
4	sendfile	/tmp/B.htm
5	send	bye

Figure 2.6: Suzumura [39] HTTP Chunked Encoding Response from Web Server

For all FTCS objects, the web server sends the file using the *sendfile()* system call (Chunks 2 and 4). Otherwise, the content is sent using a regular *send()* (Chunks 1, 3 and 5). The *sendfile()* system call copies data from one file descriptor to another within the kernel. It therefore does not incur any data copying between user and kernel space, permitting highly efficient file transfers.

To understand the performance of Suzumura's technique, let us consider two scenarios. Figure 2.7a shows the data copied when a majority of the response is static and FTCS objects are used. With the modified FastCGI protocol, only the file names of the FTCS objects are sent through the FastCGI socket, resulting in a small FastCGI response and significantly reducing the amount of data copied in C1 and C2 (Figure 2.7a, Stage 1). In Figure 2.7a, Stage 2, the web server sends each non-

FTCS object using *send()* and each FTCS object using *sendfile()*. Since *sendfile()* does not copy data between user and kernel space, only non-FTCS objects incur copying overhead C3, significantly reducing the total amount of data copied between user and kernel space when the response is sent to the client. Unfortunately, their technique does not reduce the amount of data copied for any dynamically generated response. Figure 2.7b shows the data copied when the response contains only dynamically generated data. The amount of data copied remains the same as when compared to the original FastCGI architecture.

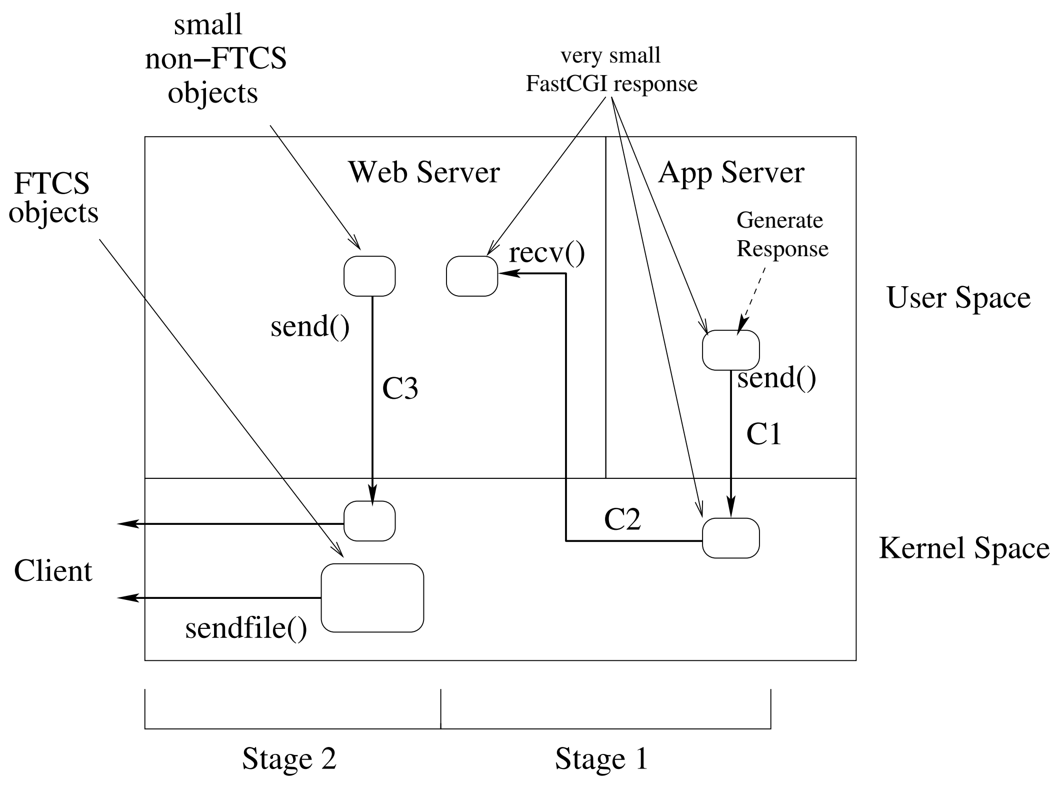
This technique using FTCS objects significantly reduces the amount of data copied between user and kernel space during response delivery when a large amount of static content is encapsulated within FTCS objects. Although this approach leverages the web server implementation of zero-copy data transfer, it requires changes to the programming language runtime library (PHP runtime library) as well as modifications to the application code (i.e., extract static content into files), which becomes a barrier to adoption. More importantly, this technique only provides performance improvements when the content being returned is stored in separate files and is included using *file_get_contents()* function. We believe that although this may be a technique used in SPECweb2005 to generate dynamic content, it is an artifact of the simple benchmark rather than a mechanism that is frequently used in dynamic content generation in real applications.

In Chapter 3, we describe our solution which eliminates two-thirds of the data copies required during response delivery. Our solution does not require modification to the application server programming language. In addition, it does not require manual encapsulation of content (such as FTCS objects and the *file_get_contents()* function) to improve performance. As a result it reduces data copying for all dynamically generated content.

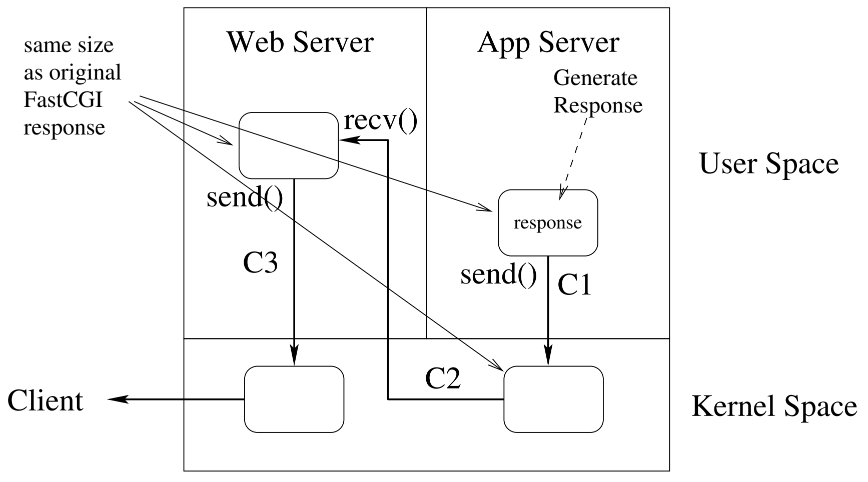
2.2.3 FastCGI with Shared Memory

An implementation of FastCGI with shared memory reduces the amount of data copied when the application server replies with dynamic content to the client. Figure 2.8 shows the different stages of the request-response cycle using FastCGI with shared memory.

The web server allocates a large shared memory region during creation of application server pool (Figure 2.8, Step 0). For each request, the web server sends a slightly modified FastCGI request to the application server. The modified FastCGI



(a) Response with Large Static Portions



(b) Response with Only Dynamically Generated Portions

Figure 2.7: Data Copying in Suzumura's Zero-Copy Technique

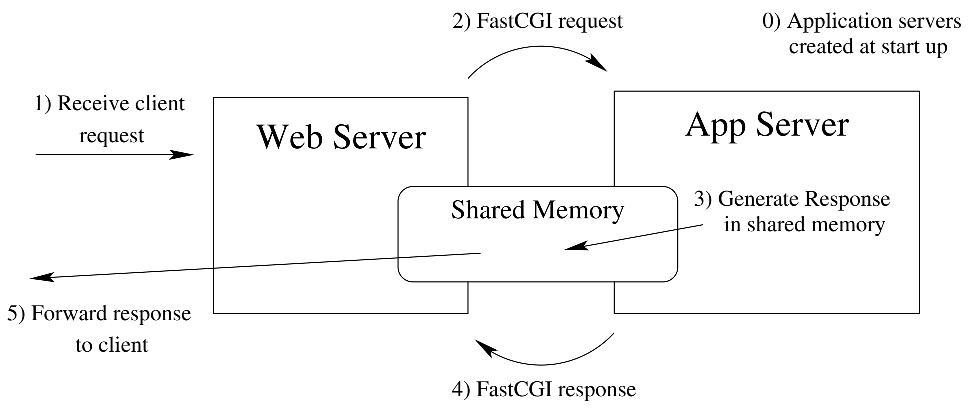
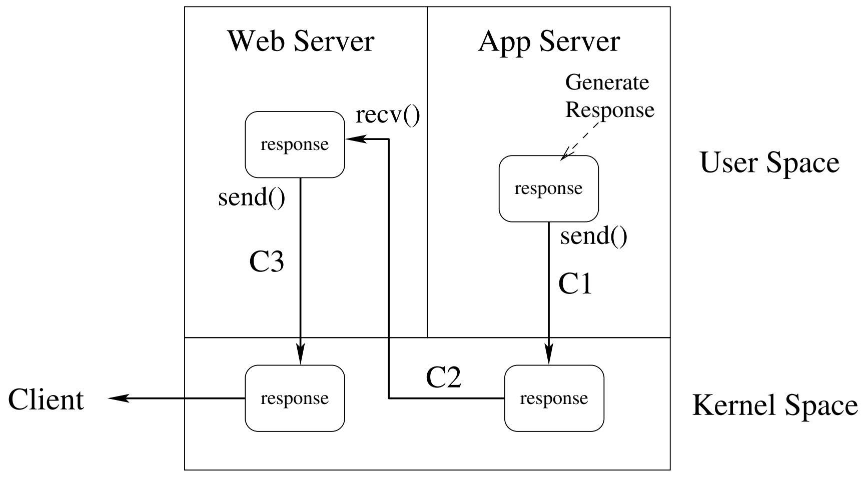

2) FastCGI request
0) Application servers created at start up
1) Receive client request
Web Server
App Server
Shared Memory
3) Generate Response in shared memory
5) Forward response to client
4) FastCGI response

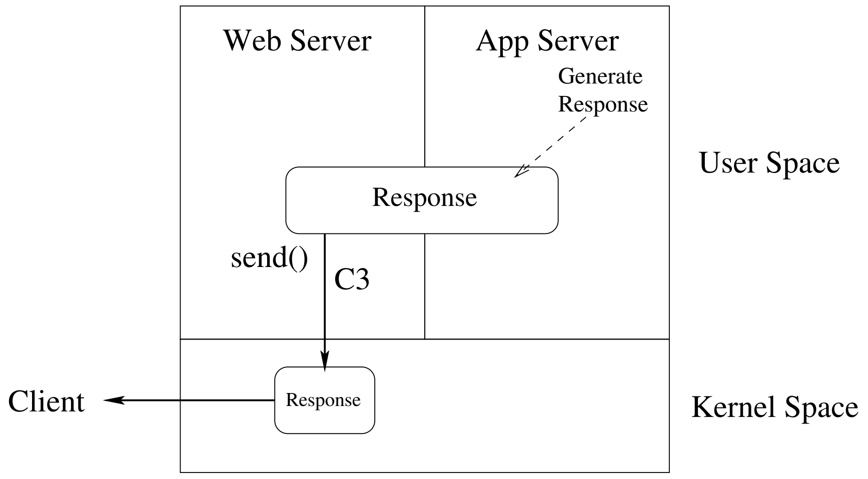
Figure 2.8: FastCGI with Shared Memory

request has two parts. First, it indicates the shared memory location that the FastCGI library should write to. Then, it is followed by the regular FastCGI request (Figure 2.8, Step 2). Instead of using a socket to send the response back to the web server, the application generates the response directly into the shared memory region (Figure 2.8, Step 3) and sends an empty FastCGI response to the web server to notify it that the response is available in the shared memory location (Figure 2.8, Step 4). Then, the web server sends the response to the client from the shared memory region (Figure 2.8, Step 5). This approach avoids copying data from the user space of an application server to a kernel buffer (which occurs during a socket write on the application server) and the copying of the data from a kernel buffer to the web server (which occurs during a socket read on the web server).

Using shared memory to transfer dynamic responses eliminates two copy operations between user and kernel space. Figure 2.9a shows the data copying operation with the original FastCGI architecture, and Figure 2.9b shows the data copying operation using FastCGI with shared memory. With FastCGI using shared memory, C1 and C2 are eliminated as the content is generated in a shared memory region. The only messages sent through the socket is the FastCGI message used to signal the web server that the response is available in the shared memory. This message size is insignificant compared to the size of the dynamic response. The only data copy happens when the web server sends the response to the client (C3) where data is copied from the shared memory buffer to the kernel. Therefore, the amount of data copied during response delivery is reduced to one-third of the amount required in the original FastCGI architecture. This technique does not reduce data copying as significantly as Suzumura's zero-copy technique when the response contains large static portions. However, if the response contains only dynamically generated content, the amount of data copied with this technique is reduced to only one-third



(a) Original FastCGI



(b) FastCGI with Shared Memory

Figure 2.9: Data Copying During Response Delivery

of the amount required by Suzumura's technique.

Normally, data in an HTTP response is delivered in one piece, with content length specified in the HTTP header. In order to determine the content length, the current proof-of-concept implementation allocates a large shared memory region that is big enough to hold the whole application server's response. The content length of the whole response can then be determined and the response is sent to the client in one piece. To improve scalability in a production environment, a smaller memory footprint may be desired. To reduce the size of the shared memory region, one could use HTTP chunked transfer encoding. Chunked transfer encoding allows data to be broken up into a series of blocks of data, and it allows the web server to transmit blocks of data to the client without knowing the content length of the entire response.

Once the application response fills up the shared memory region, the web server could use chunked transfer encoding to send the current block of data in shared memory to the client. Once the data is transferred, it would notify the application that it should resume its response, by writing again to the beginning of the shared memory region. The web server can repeat this process until the whole application response is sent.

It is possible to further reduce the amount of data copied in FastCGI with shared memory by utilizing *sendfile()* and using *mmap()* [2, 31] to map to a file. However, this enhancement is not used in our performance analysis of FastCGI with shared memory in Chapter 5 due to the lack of proper support in Linux (detailed shortly). First, the web server uses *mmap()* on a file, causing the operating system to allocate a buffer in the file cache (which resides in the kernel) and maps the buffer into the web server's address space. Then, the web server shares its memory region with the application server as described above.

When the application server generates a response into the shared memory, the response is directly written into the buffer in the file cache without any data copying between user and kernel space (Figure 2.10, Step 1). Then, the web server can use *sendfile()* to send the response back to the client (Figure 2.10, Step 2). This enhancement completely eliminates data copying between user and kernel space during response delivery. In Linux, there is currently no way to determine when the data has been sent and therefore when the buffer can be safely reused. In the future we hope to examine FreeBSD's *sendfile()* which may provide better support for such an operation.

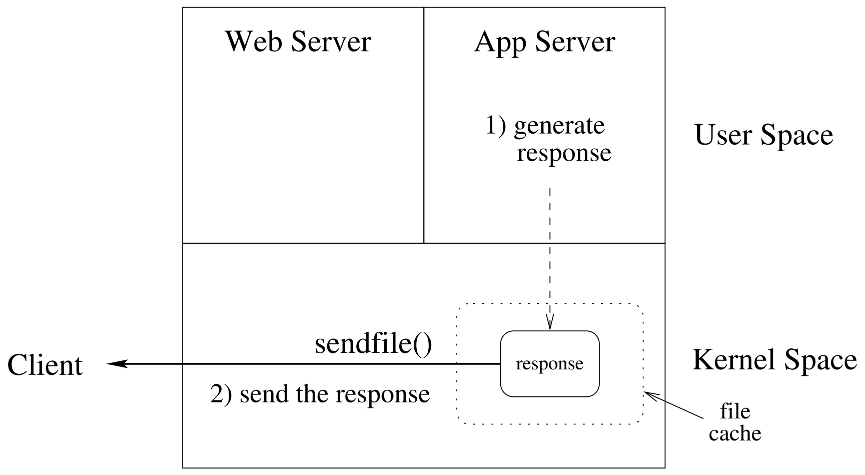


Figure 2.10: FastCGI with Shared Memory (with sendfile() and mmap() enhancements)

Therefore, FastCGI with shared memory has significant reductions in data copying during response delivery. Unlike Suzumura *et al.* [39], this optimization does not require modifications to the application server nor to the application runtime libraries. It only requires modifications to the FastCGI library and the web server. We evaluate the performance of the FastCGI with shared memory architecture in Chapter 5 and compare its performance against the original FastCGI architecture and against our new approach, as described in the next chapter.

Chapter 3

Problem, Solution and Implementation

In the previous chapter, we describe the initial development of dynamic content generation architectures, CGI and FastCGI. FastCGI was developed to improve server performance by reducing the process creation overhead of CGI. We also identified several performance problems with FastCGI and outline work related to these problems.

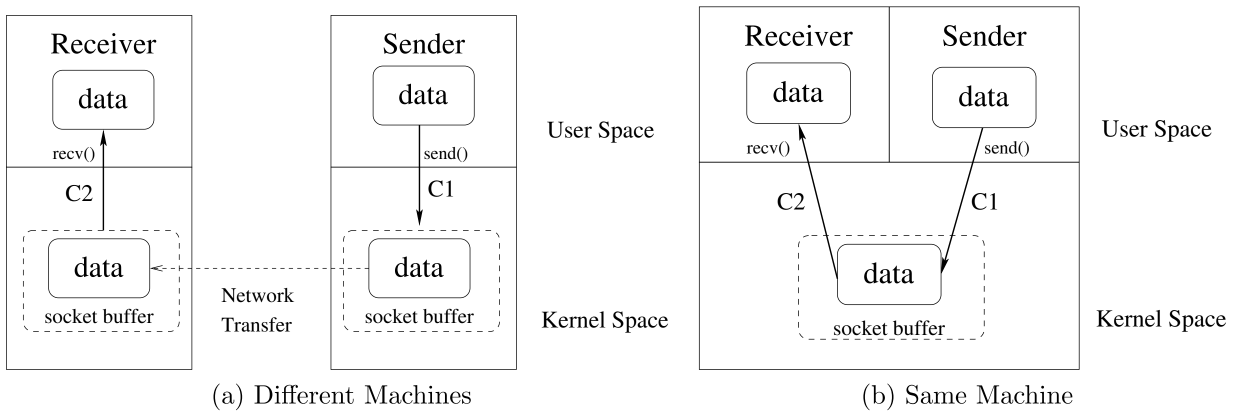
In this chapter, we focus on the redundant copying problem when delivering responses with FastCGI. First, we provide a detailed description of the problem. Second, we propose a technique that reduces copying by passing the client socket's file descriptor from the web server to the application server in order to permit the application server to respond directly to the client, thus bypassing the web server and reducing the amount of data copying. Third, we discuss the potential benefits of this technique and describe the implementation of the proposed technique. Finally, we explain how this technique can be applied to other similar systems.

3.1 Problem

In the FastCGI architecture, the response delivery phase performs two data exchanges using socket communication. When the web server and the application server are co-located on the same machine, we observe redundant data copying in the data flow. Therefore, to understand the optimization opportunity in detail, we explain when and why data copying occurs between user and kernel space when a socket is used for inter-process communication.

3.1.1 Data Copying in Socket Communication

Let us consider data copying in two steps as shown in Figure 3.1: sending to a socket and receiving from a socket. Figure 3.1a shows the socket communication when both the sender and the receiver are on different machines. First, the sender process sends data to a socket by invoking the *send()* system call. The data is copied from the sender's address space into the kernel socket buffer by the operating system (Figure 3.1a, label C1). Then, the data is sent to the destination machine by the kernel. Finally, when the receiver reads from the socket with the *recv()* system call, the data is copied from the kernel socket buffer to the receiver's address space (Figure 3.1a, label C2). Figure 3.1b shows the socket communication when the processes are on the same machine. The steps are the same as with socket communication between different machines except there is no network transfer. Both socket communications require two system calls and two data copies between user and kernel space.



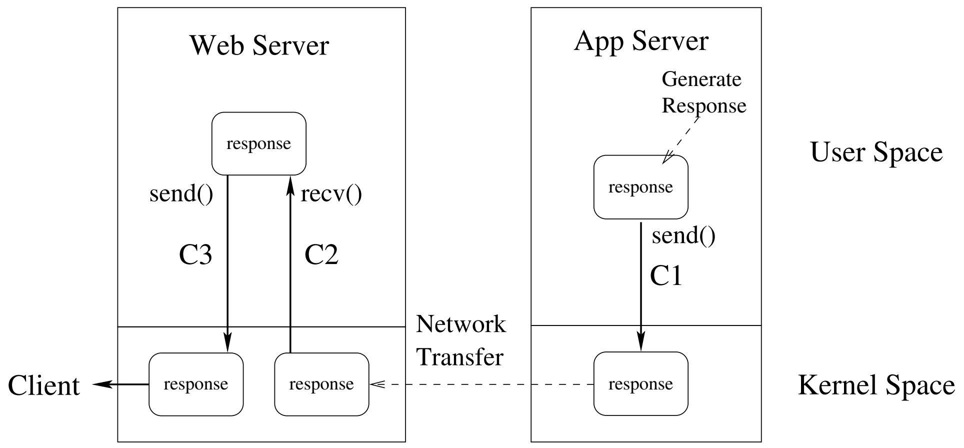
(a) Different Machines (b) Same Machine

Figure 3.1: Socket IPC Data Copying

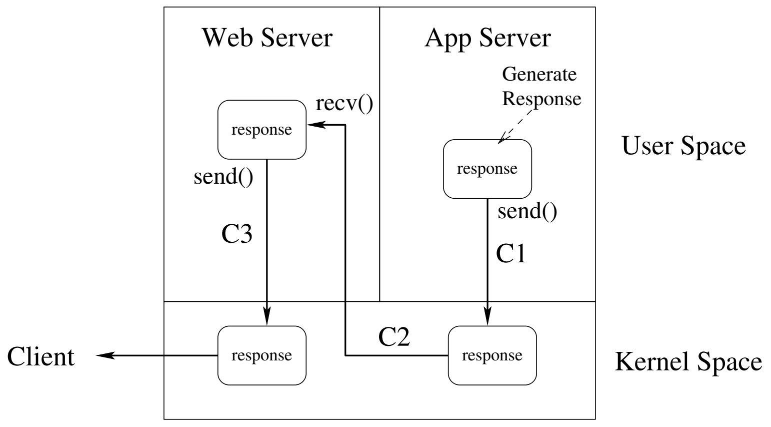
3.1.2 Data Copying in the FastCGI Architecture

The FastCGI architecture utilizes two sockets in the response cycle, the client socket and the FastCGI socket. The client socket facilitates communication between the client and the web server. The FastCGI socket facilitates communication between the web server and the application server. We focus on data copying during response delivery, because the response size can be quite large while the request size is typically very small (e.g., a few hundred bytes).

Figure 3.2a shows the response delivery when the web server and the application server are on different machines. Sending the response from the application server to the web server follows the same steps as socket communication between different machines (Figure 3.1a) and requires two data copies between user and kernel space (Figure 3.2a, labels C1 and C2). Then, the web server sends the response back to the client through the client socket causing a third copy (Figure 3.2a, label C3). Figure 3.2b shows the response delivery when the servers are on the same machine. The steps are the same as response delivery when the web server and the application server are on different machines, except there is no network transfer. Response delivery consists of three data copies between user and kernel space in both cases.



(a) Different Machines



(b) Same Machine

Figure 3.2: Data Copying in FastCGI Response Delivery

Notice that the web server simply acts as a proxy and does not perform any modification of the dynamic content, so the same data is copied between user and kernel space three times. As mentioned in the beginning of this chapter, dynamic response computation may be processor intensive. Similarly, data copying between user and kernel space in response delivery is also processor intensive. By reducing the number of copies, additional processor time is freed up. This processor time can then be used to spend more time computing responses, improving overall server performance.

3.1.3 Reducing Data Copying in the FastCGI Architecture

In Section 2.1.2, we described the economic and performance benefits of co-locating the web server and the application server on the same machine. In order to fully understand and maximize these benefits, we analyze data copying in the FastCGI architecture with the web server and the application server co-located on the same machine.

Several data copying reduction techniques take advantage of the server co-location properties. Suzumura's technique (Section 2.2.2) takes advantage of the fact that the web server and the application server share the same file system. The FastCGI with shared memory technique designed by our research group (Section 2.2.3) utilizes server co-location on the same operating system in order to share memory regions.

Our solution also utilizes the characteristics of server co-location on the same operating system. Below, we introduce our solution that passes the client socket descriptor from the web server to the application server allowing the application server to send the response directly to the client. This bypasses the web server and requires only one data copy operation between user and kernel space during response delivery.

3.2 Solution

In the problem section, we illustrated three data copies between user and kernel space in the original FastCGI architecture. In this section, we propose a technique that passes the client socket descriptor from the web server to the application server in order to reduce the number of copies during response delivery to one. Then, we describe the implementation details of our technique in the FastCGI architecture. Finally, we describe how our technique can be applied to other operating systems and web servers.

3.2.1 Passing a File Descriptor to Another Process

A file descriptor is a reference to an open file or socket. To perform actions on the file or socket, the process invokes an operating system call on the file descriptor. Then, the operating system performs operations on the file or socket on behalf of the process. Furthermore, each process has its own list of file descriptors to enforce security.

One of the ways in Linux and FreeBSD to pass an opened file descriptor to another existing process is by sending an ancillary message on a Unix domain socket. First, the two processes set up a Unix domain socket for communication (Figure 3.3a). Then, the sending process creates an ancillary message with type *SCM_RIGHTS* and sends the file descriptor through the socket. The kernel appends the underlying file or socket object pointer to the socket buffer (Figure 3.3b). When the receiving process receives this from the socket, the kernel reads from the socket buffer and decodes the pointer. Then, the kernel associates this pointer with an unused file descriptor in the receiving process (Figure 3.3c). Once this is completed, the new file descriptor is returned to the receiving process. The receiving process can now perform operations on the opened file through the new file descriptor, while the sending process can still perform operations with the original file descriptor.

Since both of the file descriptors refer to the same kernel object, they share the same offset and status flags. If the file offset is modified using *read()*, *write()* or *lseek()* on one of the descriptors, the file offset is also changed for the other file descriptor. However, file descriptor flags are not shared among the descriptors. Reference counting is applied to objects, and an object resource is freed when *close()* is called on the last reference to it. For example, suppose there are two file descriptors referring to the same TCP socket. When *close()* is called on one of the file descriptors, the reference count of the socket is decremented. However, this call does not initiate the TCP termination sequence since the reference count is greater than zero. The TCP termination sequence is initiated when the other file descriptor (the last reference to the TCP socket) is closed.

Several researchers utilized this technique to improve server performance. In Park and Pai [29], the authors use "Connection Conditioning", which decouples server-independent functions into general-purpose user-level filters to keep server design simple and increase security. Each filter passes the client's socket descriptor to the next filter. This allows filters to send data directly to the client, thus reducing the latency compared to a proxy-based solution. Ruan and Pai [34, 35] study blocking in the main server process due to lock contention in file system operations

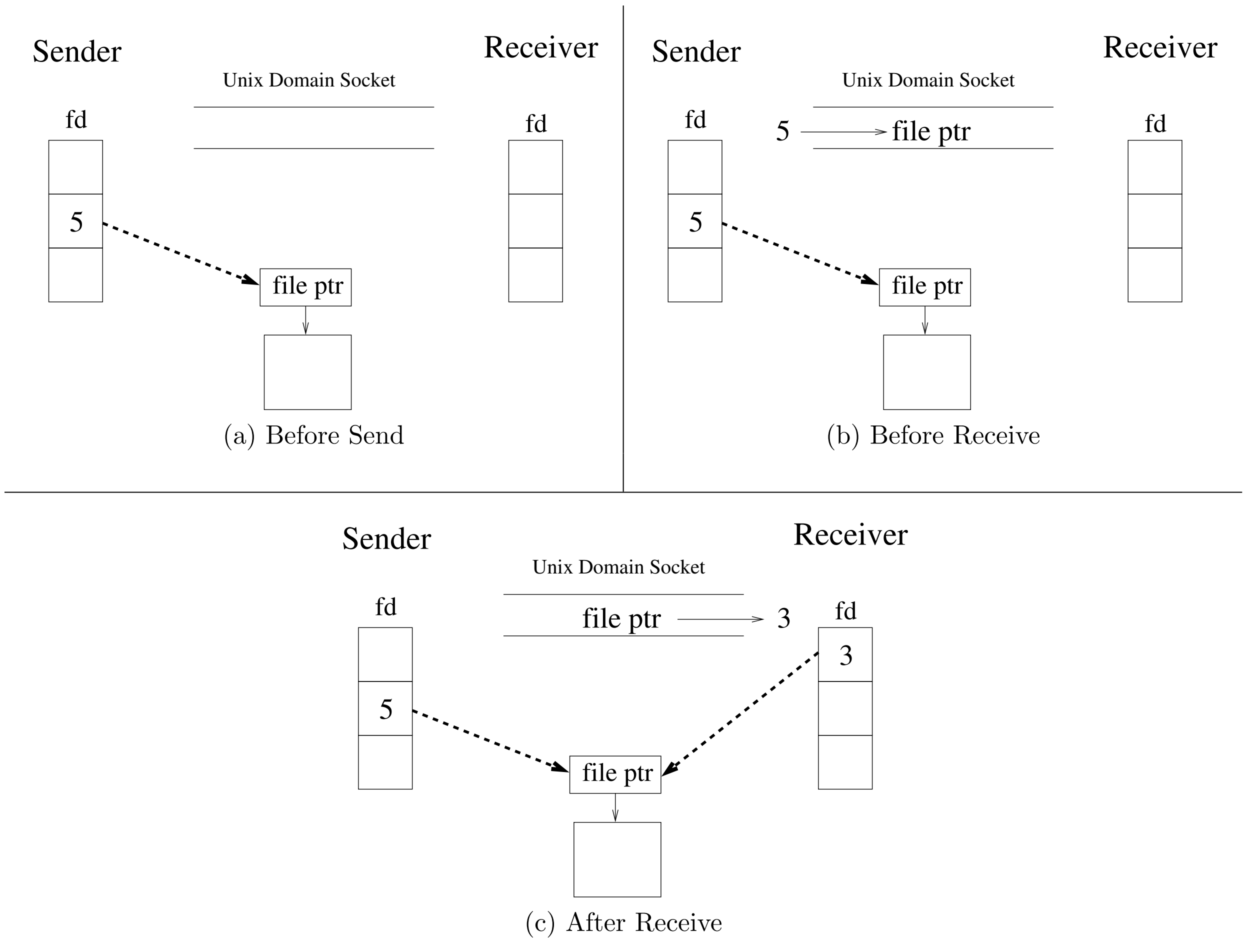


Figure 3.3: Passing a File Descriptor to Another Process

between the web server process and its worker processes. By passing file descriptors between the main server process and the worker processes, they offload work required by file system operations to the worker processes, reducing lock contention induced blocking. They implement this technique in both the Flash Web Server [42] and the Apache HTTP Server [9], increasing the throughput of the Flash Web Server by 34% and the throughput of the Apache HTTP Server by 13%.

3.2.2 FastCGI with PASSFD

In Section 3.1.2, we explained that traditional response delivery requires three data copies between user and kernel space. If the application server has a communication channel to the client, it can deliver the response to the client directly without passing the data to the web server, thus eliminating the copying done in the steps labeled C2 and C3 in Figure 3.2b. We can pass the client socket descriptor from the web server to the application server to establish a communication channel between the application and the client. We call this technique of passing the client socket descriptor from the web server to the application server, PASSFD.

Recall that the application server communicates with the web server using the FastCGI library, and that communication is encapsulated within the library. By passing the client file descriptor from the web server to the FastCGI library, a direct communication channel to the client is established and is encapsulated within the library. Since the web server does not perform any operations on the response, the FastCGI library can send the data directly to the client, bypassing the web server (Figure 3.4).

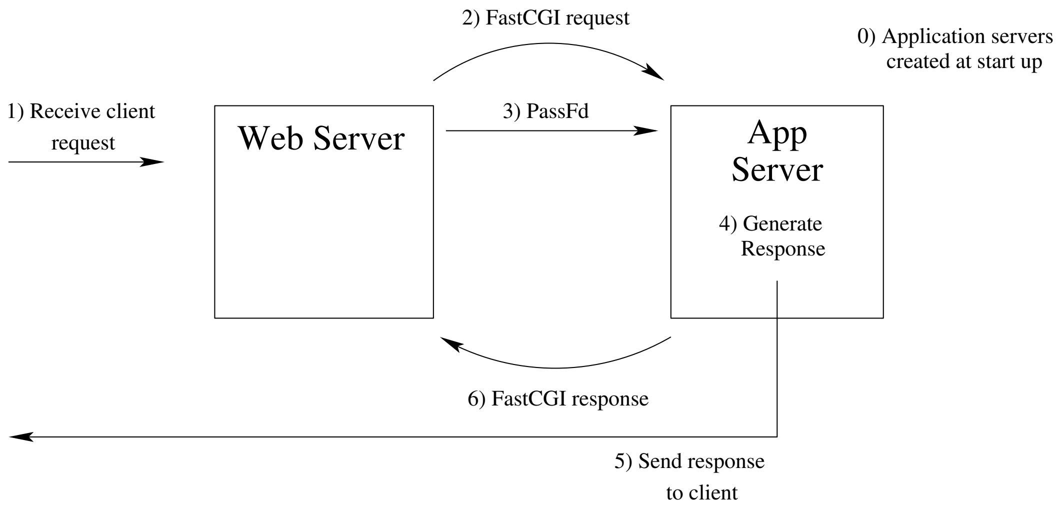

2) FastCGI request
0) Application servers created at start up
1) Receive client request
Web Server
3) PassFd
App Server
4) Generate Response
6) FastCGI response
5) Send response to client

Figure 3.4: FastCGI with PASSFD

Here is the order of events in our proposed solution. First, the web server receives a request from the client (Step 1). Then, the web server passes the client request to the application server (Step 2), at the same time it sends the client socket file descriptor along (Step 3) to the FastCGI library. The FastCGI library passes the request to the application server but maintains the client's socket descriptor. In addition, the application server code uses the *FCGX_PutStr()* function from the FastCGI library to send the response. This encapsulation allows the application server code to be completely oblivious of the receiver to whom the response is sent, and thus avoids modifying the application server code in any way. When the application finishes generating its response, the library prepares the HTTP header (portions of which is also obtained from the web server along with the client file descriptor) and sends the header and response directly to the client (Step 5). Afterwards, the FastCGI library sends a FastCGI response with an empty data payload to the web server to complete the response delivery (Step 6). The web server receives the FastCGI response, skips the sending of the response to the client and instead waits for another request from the client.

We call this solution FastCGI with PASSFD. Figure 3.5 shows the data copied between user and kernel space during response delivery with this technique. The response bypasses the web server and is sent directly to the client. Therefore, there is only one data copy (Figure 3.5, C1). When compared with the original FastCGI architecture using Unix domain sockets, this technique reduces the amount of data copied by two-thirds, which is the same amount of data copy required by FastCGI with shared memory.

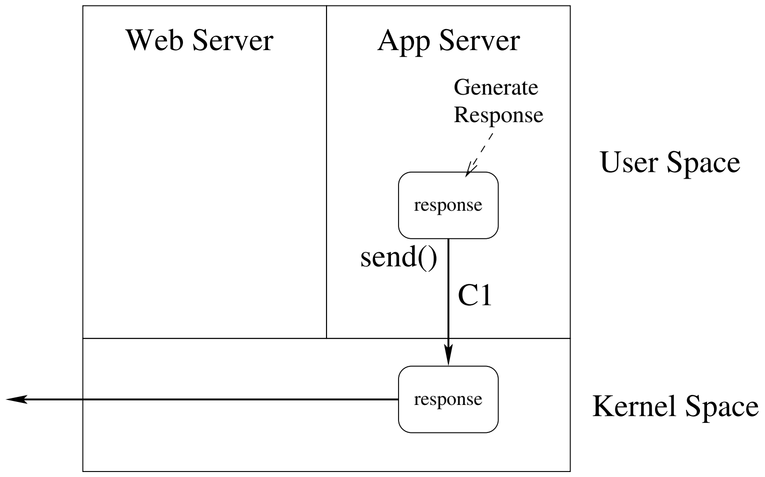


Figure 3.5: Data Copying During Response Delivery with FastCGI and PASSFD

We conduct experiments in Chapter 5 to understand how the reduced copying translates to extra server throughput, and we compare the performance of our solution to the original FastCGI architecture using Internet sockets, the original FastCGI architecture with Unix domain sockets and FastCGI with shared memory.

3.3 Implementation Details

The implementation of the PASSFD technique in the FastCGI architecture requires modification to the web server and the FastCGI library. As explained above, this solution avoids modifying either the application server or the runtime language library. Furthermore, this technique reduces the amount of data copied even for dynamic content, unlike Suzumura's technique. Next, we describe the changes required in server initialization, request handling, response handling and error conditions.

3.3.1 Changes to Server Initialization

To avoid any possibility of introducing errors into the regular FastCGI protocol, in our prototype implementation we create an extra Unix domain socket dedicated to passing file descriptors from the web server to the application server. To distinguish among the different sockets, we call this the PASSFD socket. We call the original socket used for FastCGI communications between the web server and the application server the FCGI socket. In a production environment, all communications between the web server and application servers could use the same socket. Finally, we call the socket between the web server and the client, the client socket.

During server initialization, after the application servers are created, the web server first sends a FastCGI control message over the FCGI socket, so the FastCGI library knows to use the PASSFD mode. Then, the FastCGI library creates the PASSFD socket and waits for the web server to connect to it. This initialization happens before the web server receives any requests. Regular initialization continues once the PASSFD socket connection is established.

3.3.2 Changes to Request Delivery

In regular FastCGI request delivery, the web server sends the client request to the application. In FastCGI with PASSFD, after the web server sends the FastCGI request using the FCGI socket, it passes the client socket file descriptor to the FastCGI library through the PASSFD socket. The FastCGI library first reads the client request off the FCGI socket. Then, it performs an extra receive on the PASSFD socket to retrieve the client socket file descriptor. Once the client socket is received, the FastCGI library uses it to deliver the request to the application.

3.3.3 Changes to Response Delivery

Application servers use the *FCGX_PutStr()* function to pass the response to the FastCGI library. The FastCGI library uses a response buffer to hold the dynamically generated response. In the original FastCGI architecture, the FastCGI library sends partial responses back to the web server when this buffer is full. The web server is responsible for buffering the partial responses and sending them to the client when the response generation is complete.

However, in our solution, the FastCGI library sends the response back to the client directly without the web server acting as an intermediate buffer. To handle partial responses when the FastCGI response buffer is full, we implement HTTP standard chunked encoding transfers [8] in the FastCGI library. HTTP chunked encoding transfers allow HTTP responses to be sent in chunks without specifying the full response length in the HTTP header. The end of the response is indicated with a zero sized chunk. So, when the response buffer is full, the FastCGI library sends the chunk to the client. The client then assembles these chunks together to form a complete response.

When the full response has been sent to the client, the FastCGI library first closes the client socket. Then, it replies to the web server with a FastCGI response message with an empty data payload. The web server recognizes an empty data payload and either terminates the client connection or waits for the next client request according to whether or not the client is using a persistent connection. Even though the FastCGI library closes the client socket, due to reference counting in the kernel, the client connection remains connected until the web server terminates the client connection.

Since the application server sends the response to the client directly, it also prepares and sends the HTTP header. In our experimental prototype, the application server sends a header with the response code 200 and HTTP chunked encoding transfer enabled. In production environments, we need to determine if all HTTP header options can be prepared by the application server. If there are specific HTTP headers that must be sent by the web server, the web server can set TCP_CORK [13, 21] in Linux or TCP_NOPUSH [30] in FreeBSD on the client socket and send the HTTP headers. TCP then delays sending any data until the response is sent by the application server.

3.3.4 Handling Error Conditions

If the client terminates prematurely (e.g., in the case of timeout), the FastCGI library receives an error when it sends the response to the client. The library then stops sending data to the client. The application server proceeds normally until the end of the response has been generated. No special handling is required as the error condition is detected by the web server when it performs further actions on the client connection (i.e., closing or reading from the connection). Unfortunately, no special action can be taken to indicate to the application server that it does not need to continue generating a response without modifying the application code, so there can be extra effort wasted on timed out clients.

3.4 Applying PASSFD to Other Systems

In this chapter, we describe our solution and discuss its implementation using the Linux operating system interface and FastCGI architecture as a proof-of-concept. PASSFD is applicable to a variety of operating systems and server architectures. It only requires an operating system that supports passing of the client socket, along with a web server and an application server that supports FastCGI.

3.4.1 Operating Systems that Can Pass File Descriptors

Similar to our implementation in Linux, we can implement PASSFD in FreeBSD, OpenBSD and OSX by sending and receiving an ancillary message with *SCM_RIGHTS* over a Unix domain socket connecting the sender and the receiver. In Solaris, we can implement PASSFD by calling *ioctl()* on a named stream pipe [38]. On Windows, the *WSADuplicateSocket()* function in the Winsock library facilitates file descriptor passing between processes [15, 18].

3.4.2 Modifications to the Web Server

Any web server that implements the FastCGI protocol can be modified to implement PASSFD. As described in Section 3.3, we modify the server initialization to create the new PASSFD socket. Then, we modify request delivery to pass the client socket to the FastCGI library. Lastly, we change the response delivery stage to skip writing the response to the client. The number of lines of code changed in our web server is less than 500, and these modifications apply to both event-driven web servers as well as thread-based web servers.

3.4.3 Modifications to the FastCGI Library

Modifications to the FastCGI library are similar to the modifications to the web server. We modify library initialization to create the new PASSFD socket. Then, we modify the receive request code to receive the client socket from the PASSFD socket. We also modify the library to send the response over the CLIENT socket when the response buffer is full or when the response generation is complete. In total, we have modified around 300 lines of the library code, which includes the new code added to use HTTP chunked encoding transfers. Note that these are one time changes that can be utilized in conjunction with any web server that has been modified to support these new features. Most importantly, no changes are required to the application server code.

Our solution can be implemented on the Microsoft Windows operating system with the Apache HTTP server, FastCGI library and IBM WebSphere Application server using the *WSADuplicateSocket()* function to implement PASSFD. In general, PASSFD can be adapted to servers that delegate tasks to other processes using inter-process communication when the response can bypass the delegator.

Chapter 4

Experimental Setup

In Chapter 3, we describe opportunities to reduce the amount of data copied in the FastCGI architecture when the web server and the application server are co-located on the same machine. This reduction can be accomplished by either sharing memory between the application server and the web server, or implementing PASSFD in the FastCGI library and the web server.

Our next goal is to evaluate the performance of these techniques as well as to understand the amount of time the processor spends on copying data. The experiments we developed to achieve these goals are described in this section.

First, we describe the design of our micro-benchmark workload and discuss important decisions made in choosing the workload. Then, we describe the hardware and operating system configurations. Finally, we describe the four FastCGI architectures of interest in this study, develop the experimental plan, and discuss the performance measures.

4.1 Experimental Workload

One of the goals of our performance analysis is to understand the overhead incurred due to data copying during response delivery. The experimental workload is designed specifically to generate significant data copying overhead and allow us to determine the amount of time the processor spends copying data.

4.1.1 Response Size

Inter-process communication includes a static overhead that is independent of the size of the data being communicated, such as the system call overhead. It also includes an overhead that is proportional to the amount of data transferred. As Web 2.0 technologies expand the variety of Internet applications, response size varies under different workloads. For example, the average response size of the e-commerce scenario in SPECWeb2009 is 143.9 KB [6], whereas the file size of a 500-word MP3 audio file generated by the iSpeech [16] text-to-speech service is 1 MB. To achieve a thorough understanding of the data copying overheads and to analyze the performance of the architectures under workloads with different response sizes, we construct workloads with different response sizes. The response sizes are 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, and 4 MB with each workload using one response size.

To stress the computational capacity of the server, we have the clients issue a single request which reads the content of a file and performs some simple transformations on the file content. These are identical to those done in the SPECWeb99 benchmark [4]. The entire file easily fits in memory and requires no disk I/O.

4.1.2 Client Timeout

A client timeout is defined as the amount of time a client waits for a server response before terminating its HTTP connection or the amount of time the browser software waits before closing the connection. Any responses sent by the server after the client times out are ignored by the client. Since we would like to compare the amount of processor resources used in data copying, we would need to account for failed responses, quantify the amount of CPU wasted and normalize the amount of processor resources used in data copying for fair comparisons. This accounting task would be complex and very likely inaccurate. To understand the effect of client timeouts, consider an experiment using a small client timeout value. Suppose the server is busy and a connection times out after the request has been passed to the application server. In this scenario, subsequent computation by the application server and communication overhead are wasted.

Instead, we avoid this situation by eliminating client timeouts. Without client timeouts, the workload generator does not generate overload conditions because it only sends a new request after the server has replied to its previous request. Research has shown [1] that the lack of connection timeout causes the client's request rate to be throttled by the reply rate of the server. Therefore, the rate of client requests cannot exceed the peak service rate of the server. This behaviour prevents the system from experiencing overload conditions. Hence, we did not analyze system dynamics after the server saturates. Rather, we focus our data copying bottleneck analysis on loads up to the point of server saturation.

4.2 Experimental Environment

Our experimental environment consists of four client machines and one server machine. We have two network setups for our experimental traffic. One utilizes four, 1 Gbps subnets, and the other utilizes eight, 1 Gbps subnets. Both networks use an additional 100 Mbps subnet to send control commands.

4.2.1 Client Machine Setup

Each client machine contains two hyper-threaded Intel(R) Xeon(TM) 2.40 GHz processors, 1 GB of RAM, 1 GB of swap space and four Intel(R) 1 Gbps ethernet cards. They are all running version 2.4.20-8 of the Linux kernel with SMP enabled.

We use *httperf* version 0.8.4 [25] with minor local modifications to drive the workload. *httperf* is an event-driven workload generator that avoids thread management overhead [25]. We increase the maximum limit on the number of open files per process from 1024 to 65535 to prevent *httperf* from running out of file descriptors.

4.2.2 Server Machine Setup

The server has four Intel(R) Itanium 2(TM) 1.50 GHz processors, 8 GB of physical memory and eight Intel(R) e1000 1 Gbps ethernet cards. It runs a 2.6.9 Linux kernel. We use the *µserver* web server [3, 28], version 2.4.0 of the FastCGI library and an application server written in C. The application server is ported from an implementation in the SPECweb99 distribution [4]. It reads the contents of the file requested, performs some simple transformations on the contents that simulates advertisement rotations, and sends them to the client. This environment mimics web servers using dynamic scripts to generate content for advertisement on web pages in real time, so that the advertisement space can be sold to different customers. For FastCGI with shared memory and FastCGI with PASSFD, we use the *userver* and FastCGI library with modifications specific to each technique.

4.2.3 Network Setup

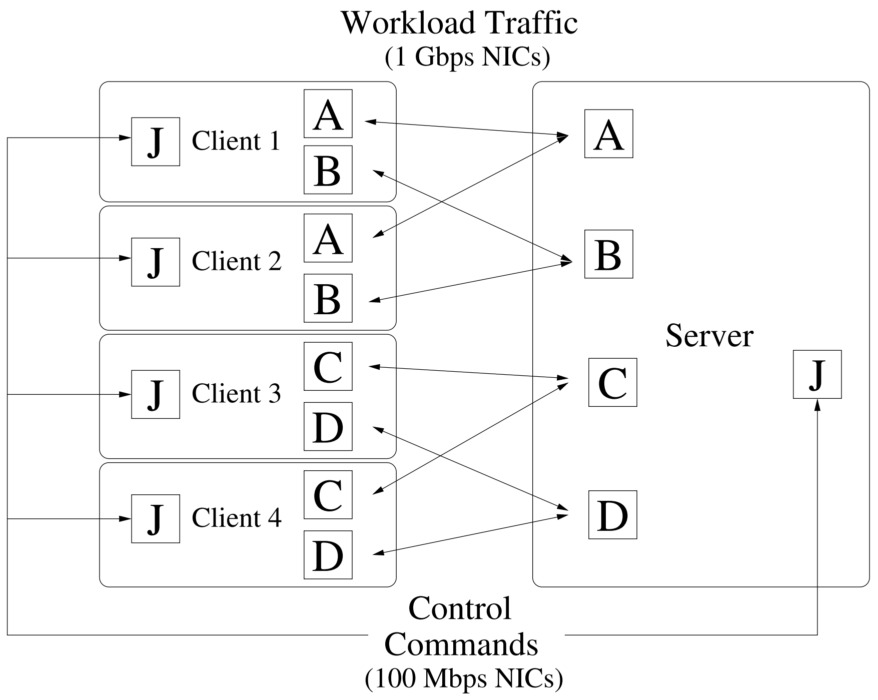
We use two different network setups in our experiments. The first one uses four x 1 Gbps subnets and the second uses eight x 1 Gbps subnets. Figure 4.1a shows the first network setup and demonstrates the separation of workload traffic and control commands. We partition the clients' gigabit network interfaces into 4 different subnets. Client machines 1 and 2 connect to subnets A and B, whereas client machines 3 and 4 connect to subnets C and D. The web server listens to requests coming from the four x 1 Gbps subnets (A, B, C, D). All of the machines are connected to an administration subnet J with a 100 Mbps network interface to isolate control commands from consuming bandwidth of the four subnets that perform the actual work. Figure 4.1b shows the second setup. It is very similar to the first setup, except we have eight x 1 Gbps subnets and each of the 4 clients connects to two of the 8 subnets.

Measuring the TCP network bandwidth with the *Iperf* bandwidth measurement tool [40], each subnet has a network bandwidth of around 940 Mbps. With both the first setup and the second setup, we can conduct experiments with network throughput of up to 3760 Mbps and 7520 Mbps respectively without reaching any network bottlenecks. All machines are connected via 24-port full duplex non-blocking gigabit switches.

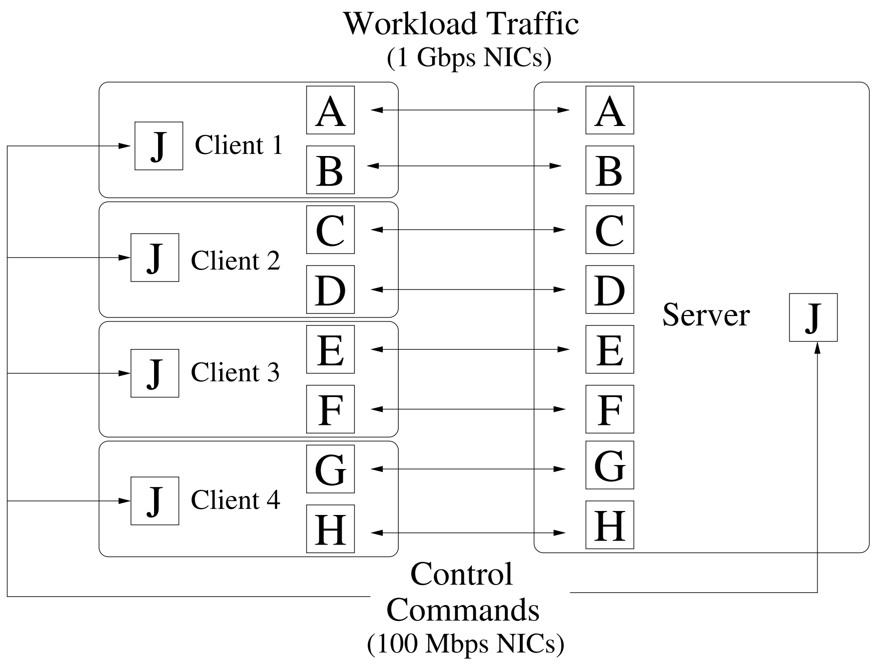
4.3 Server Architectures and Configurations

In this section, we briefly review the four FastCGI architectures used in this study (Table 4.1). Then, we describe the four server configurations that we use with each of the FastCGI architectures.

The first two FastCGI architectures (fcgi_inet and base_uds) are the original FastCGI architectures. They differ in the type of socket used between the web server and the application. FastCGI with Internet (fcgi_inet) sockets uses Internet sockets (which are required between two different machines), and FastCGI with Unix domain sockets (base_uds) uses Unix domain sockets (which can be used between processes on the same machine). We include FastCGI with Internet sockets



(a) Network Setup 1 with 4 Subnets



(b) Network Setup 2 with 8 Subnets

Figure 4.1: Network Setup

to understand the performance difference between the two types of sockets and to motivate the use of Unix domain sockets when the web server and application

Name	FastCGI Architecture
fcgi_inet	FastCGI with Internet sockets (AF_INET)
base_uds	FastCGI with Unix domain sockets (AF_UNIX)
sharedmem	FastCGI with shared memory
passfd	FastCGI with PASSFD

Table 4.1: FastCGI Architectures

servers are co-located on the same machine.

FastCGI with shared memory and our solution, FastCGI with PASSFD, both perform one data copy between user and kernel space in response delivery. We would like to compare their performance and understand the factors that may cause performance differences between them.

Finally, we would like to compare the performance of FastCGI with Unix domain sockets against both FastCGI with shared memory and FastCGI with PASSFD to understand the performance impact of reducing data copying in dynamic response delivery.

For each of the above architectures, we conduct experiments with the following configurations (shown in Table 4.2). Each column of the table lists a configuration parameter and each row represents a server configuration. To understand the change in performance when we increase the number of processors, we keep the same number of web servers and application servers while increasing the number of processors. Configurations 1 CPU 2U and 2 CPU 2U both have two web servers and 160 application servers, and we increase the number of processors from one (in Configuration 1 CPU 2U) to two (in Configuration 2 CPU 2U). We have a similar setup for Configurations 2 CPU 4U and 4 CPU 4U. Both of them have 4 web servers and 320 application servers, and we increase the number of processors from two (in Configuration 2 CPU 4U) to four (in Configuration 4 CPU 4U). By comparing the performance between Configurations 1 CPU 2U and 2 CPU 2U, and between Configurations 2 CPU 4U and 4 CPU 4U, we can understand the change in performance of each architecture with respect to an increase in the number of processors. The number of web servers and the number of application servers are chosen from basic tunings to ensure the processors are fully utilized and performance is not limited by the availability of application servers.

Configurations 1 CPU 2U, 2 CPU 2U and 2 CPU 4U use network setup 1 (shown in Figure 4.1a) and Configuration 4 CPU 4U uses network setup 2 (shown in Figure 4.1b). We would like to understand the performance of each architecture under different server configurations, in particular, different number of processors.

Configuration	Number of CPU	NICs per CPU	Number of Web Servers	Applications per Web Server	Total Number of Applications	Network Setup
1 CPU 2U	1	4	2	80	160	1
2 CPU 2U	2	2	2	80	160	1
2 CPU 4U	2	2	4	80	320	1
4 CPU 4U	4	2	4	80	320	2

Table 4.2: Experimental Server Configuration

To promote effective processor cache usage, we improve processor affinity by pinning the web server, its associated applications and its associated network interfaces to the same processor. For example, in Figure 4.2, we show a visualization of the 2 CPU 2U configuration. One web server is pinned to one of the processors (CPU0) with its 80 application servers and two network interrupt handlers (INT0 and INT1), while the other web server is pinned to the other processor (CPU1) with its application servers and network interrupt handlers (INT2 and INT3).

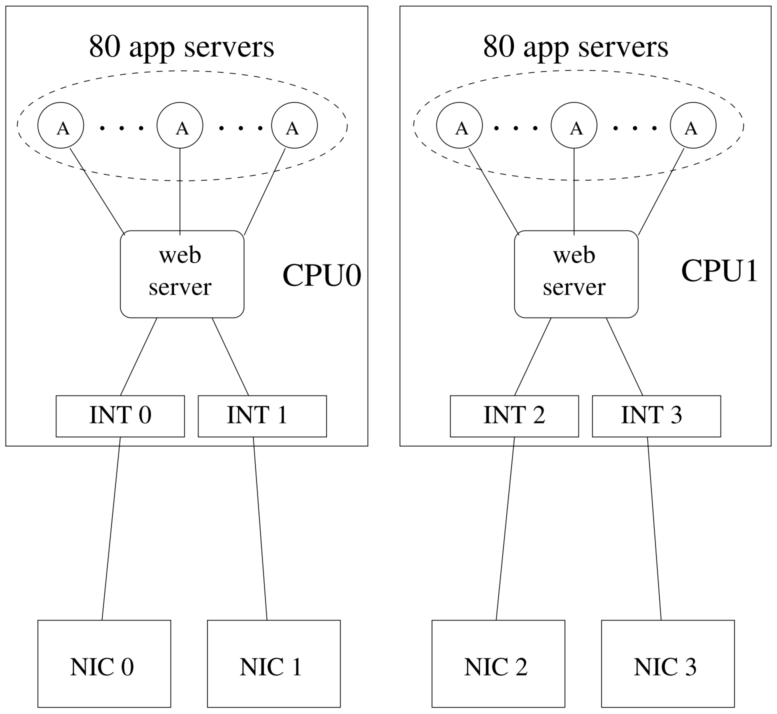


Figure 4.2: Processor Pinning in Configuration 2 CPU 2U

4.4 Experimental Methodology

We use two sets of experiments to evaluate the performance of all architectures while keeping the number of experiment runs reasonable. The first set of experiments uses the 2 CPU 2U configuration in Table 4.2 and varies the response size of the workload to evaluate peak server throughput across different response sizes.

In Section 5.1.1, the experimental results show that the performance of FastCGI with PASSFD and FastCGI with shared memory relative to that of the base case is fairly stable across different response sizes greater than 128 KB. Since performance is fairly stable across different response sizes greater than 128 KB, we do not need to consider all response sizes in the second set of experiments. The second set of experiments uses a workload with 1 MB response size and varies the server configurations to evaluate peak server throughput under different server configurations.

4.5 Performance Measurement

When initial requests arrive at an empty system, performance is likely different from a system in its steady state. To ensure capturing accurate results, we enforce a minimum experiment duration of 300 seconds to allow the system to reach steady state before collecting measurements. *httperf* uses an HTTP trace file to generate the experimental workload. If an HTTP trace takes less than 300 seconds to complete, the clients repeat the trace from the beginning until the minimum duration is satisfied.

We run *vmstat* and *mpstat* to sample statistics on the server every 5 seconds. Since they only sample every 5 seconds, they do not consume significant resources. Additionally, to capture processor time, we use *OProfile* 0.8.1 [17, 10]. *OProfile* is a low overhead system-wide profiler that leverages hardware performance counters in the processor to allow profiling all code including interrupt handlers, kernel functions, shared libraries and application code. In our analysis, we use *OProfile* to capture the percentage of processor time spent in each function and later the number of cache misses of each function. Then, we use our own scripts to organize and analyze the results. We perform experiments to determine the effects of *OProfile* overhead on the peak throughput. Table 4.3 shows the peak throughput without profiling relative to the peak throughput with profiling turned on. The peak throughput is 1.2% to 2.3% higher with *OProfile* turned off. We believe this to be small enough that all experiments are conducted with *OProfile* turned on, which eliminates the need to repeat experiments in order to perform detailed analysis of the results.

Configuration	Architecture	Response Size	Relative Performance
2 CPU 2U	base_uds	512 KB	1.023
2 CPU 2U	base_uds	1 MB	1.023
2 CPU 2U	passfd	512 KB	1.012
2 CPU 2U	passfd	1 MB	1.017
4 CPU 4U	base_uds	1 MB	1.018

Table 4.3: Oprofile Overhead

Chapter 5

Performance Analysis

In this chapter, we conduct a performance analysis of the previously described FastCGI architectures to understand the cost of data copying redundancy during response delivery. Our analysis focuses on two performance measures: peak throughput and copying overhead.

First, we compare the peak throughput achieved using each of the architectures to present an overview of the performance differences among the architectures. Then, we examine copying costs.We analyze these performance measures just prior to server saturation, that is at their peak throughput.

Later in the chapter, we look at processor cache hit statistics to investigate the performance differences between FastCGI with shared memory and FastCGI with PASSFD, as well as to understand how and why copying costs increase near saturation. Finally, we discuss the performance of our solution under a realistic workload with reference to the performance study by Suzumura *et al.* [39].

5.1 Peak Throughput

In this section, we compare the peak throughput of the various architectures using FastCGI with Unix domain sockets as the base case. Peak throughput is measured as the maximum data throughput achieved for that particular architecture under a workload of the specified response size and server configuration.

5.1.1 Peak Throughput with Different Response Sizes

This set of experiments evaluates the peak throughput of all architectures under workloads with different response sizes. As mentioned in Section 4.4, this set of experiments is run with two processors, two web servers and a total of 160 application servers (Configuration 2 CPU 2U in Table 4.2). We discuss the peak throughput of the other configurations in Section 5.1.2.

Figure 5.1 shows the peak throughput of all architectures as the response size changes. The y-axis shows the peak data throughput in Mbps. Each cluster of bars shows the peak throughput of the architectures under a workload with the labeled response size. FastCGI with PASSFD has the highest peak throughput over all the response sizes, FastCGI with shared memory has the second highest peak throughput followed by FastCGI with Unix domain sockets and FastCGI with Internet sockets.

Table 5.1 lists the peak throughput and performance relative to FastCGI with Unix domain sockets. The first column lists the FastCGI architecture, and the second column lists the response size used in the workload. The third column and the fourth column list the peak throughput and the performance relative to the base case, respectively. The performance relative to FastCGI with Unix domain sockets (the base case) is calculated as the peak throughput of the architecture divided by the peak throughput of FastCGI with Unix domain sockets.

With response sizes greater than 128 KB, the performance of FastCGI with PASSFD is 1.26 to 1.31 times that of the base case, and the performance of FastCGI with shared memory is 1.13 to 1.16 times that of the base case. FastCGI with Internet sockets has a performance of 0.88 to 0.98 times that of the base case (i.e., it has a lower peak throughput). In each architecture, the performance relative to the base case is fairly stable across larger response sizes (above 128 KB).

With a response size of 32 KB, FastCGI with PASSFD has a peak throughput of 1.06 times that of the base case, but FastCGI with shared memory does not show performance improvement. With a response size of 64 KB, FastCGI with PASSFD has a peak throughput of 1.15 times that of the base case. FastCGI with shared memory has a peak throughput of 1.11 times that of the base case. This small difference is due to less copying overhead when the response size is small. Since our approach is designed to reduce copying overheads, we focus on larger response sizes (above 128 KB), where copying cost becomes significant.

As is shown in Figure 5.1 and Table 5.1, FastCGI with Internet sockets performs

Architecture	Response Size	Peak Throughput (Mbps)	Performance Relative to Base Case
fcgi_inet	32 KB	1964.4	0.92
fcgi_inet	64 KB	2481.6	0.96
fcgi_inet	128 KB	2420.8	0.92
fcgi_inet	256 KB	2521.6	0.92
fcgi_inet	512 KB	2519.2	0.98
fcgi_inet	1 MB	2476.0	0.92
fcgi_inet	2 MB	2206.4	0.88
fcgi_inet	4 MB	2182.3	0.93
base_uds	32 KB	2140.8	1.00
base_uds	64 KB	2587.3	1.00
base_uds	128 KB	2631.2	1.00
base_uds	256 KB	2731.2	1.00
base_uds	512 KB	2574.5	1.00
base_uds	1 MB	2685.6	1.00
base_uds	2 MB	2517.4	1.00
base_uds	4 MB	2350.2	1.00
sharedmem	32 KB	2207.2	1.03
sharedmem	64 KB	2877.6	1.11
sharedmem	128 KB	3052.0	1.16
sharedmem	256 KB	3151.2	1.15
sharedmem	512 KB	2938.7	1.14
sharedmem	1 MB	3102.7	1.16
sharedmem	2 MB	2856.1	1.13
sharedmem	4 MB	2706.0	1.15
passfd	32 KB	2260.8	1.06
passfd	64 KB	2984.0	1.15
passfd	128 KB	3367.7	1.28
passfd	256 KB	3571.0	1.31
passfd	512 KB	3358.4	1.30
passfd	1 MB	3523.2	1.31
passfd	2 MB	3161.7	1.26
passfd	4 MB	3011.4	1.28

Table 5.1: Peak Throughput with Varying Response Size

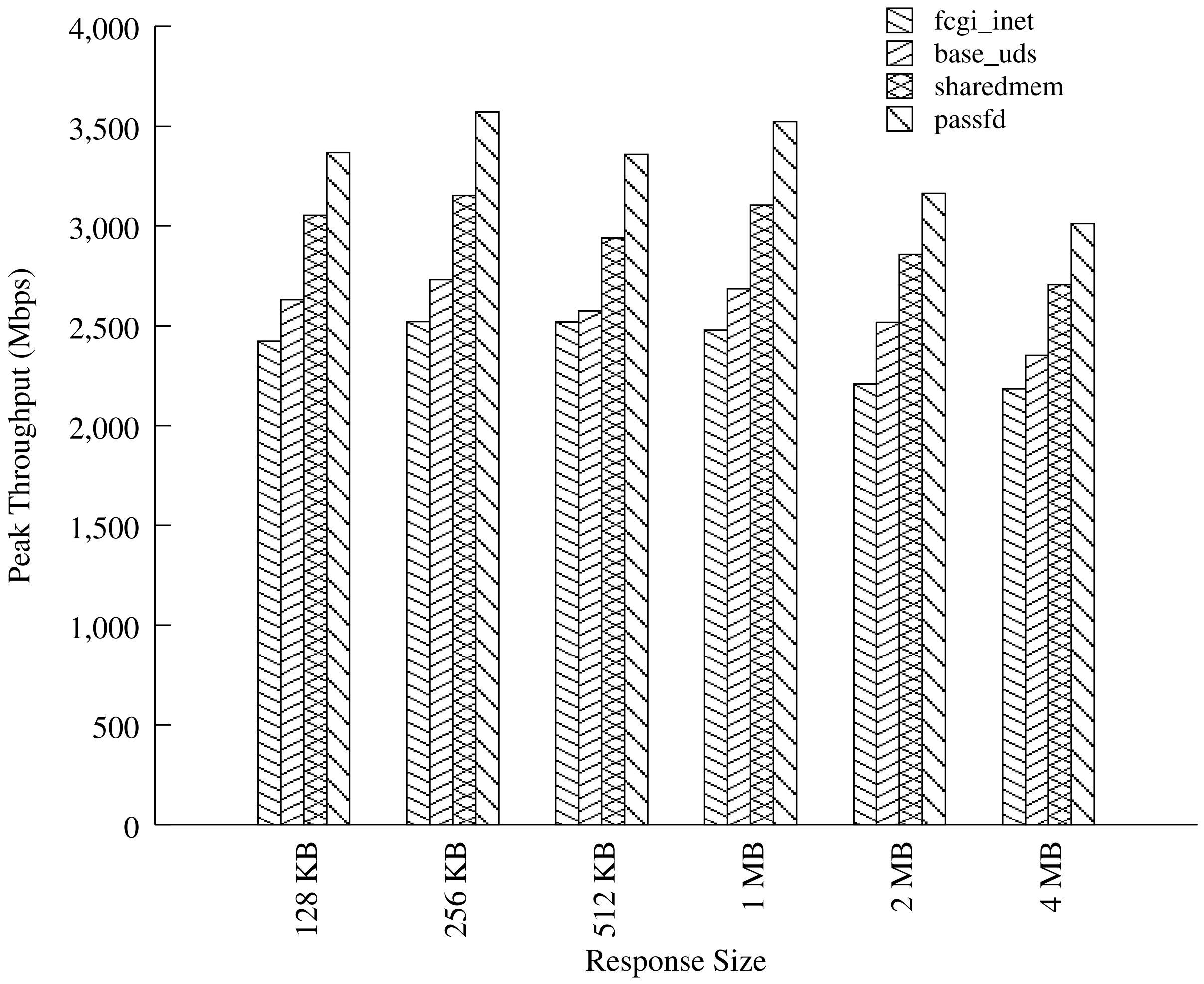


Figure 5.1: Peak Throughput with Different Response Sizes

worse than FastCGI with Unix domain sockets. Therefore, we believe FastCGI with Unix domain sockets should be used when the web server and the application server are co-located in the same machine. Since FastCGI with shared memory and FastCGI with PASSFD have quite consistent performance improvements over FastCGI with Unix domain sockets across all sizes above 128 KB, there is no need to consider all response sizes in the next sections and we focus on a 1 MB response size.

5.1.2 Peak Throughput under Different Server Configurations

This set of experiments investigates the performance of FastCGI architectures under different server configurations.

Table 5.2 and Figure 5.2 show the peak throughput of all architectures and the associated improvement with respect to FastCGI with Unix domain sockets under the same server configuration. The first and second columns of Table 5.2 list the server configuration and the FastCGI architecture used to conduct the experiment. The third column is the peak throughput captured for the server configuration and architecture pair. The last column shows the peak performance of an architecture relative to the peak performance of the base case under the same configuration. For example, in the 1 CPU 2U configuration, FastCGI with PASSFD has a peak throughput of 2305.3 Mbps and FastCGI with Unix domain sockets has a peak throughput of 1848.1 Mbps. So FastCGI with PASSFD has a peak throughput of 1.25 times the peak throughput of FastCGI with Unix domain socket.

Configuration	Architecture	Peak Throughput (Mbps)	Performance Relative to Base Case
1 CPU 2U	fcgi_inet	1635.6	0.89
1 CPU 2U	base_uds	1848.1	1.00
1 CPU 2U	sharedmem	2116.1	1.15
1 CPU 2U	passfd	2305.3	1.25
2 CPU 2U	fcgi_inet	2476.0	0.92
2 CPU 2U	base_uds	2685.6	1.00
2 CPU 2U	sharedmem	3102.7	1.16
2 CPU 2U	passfd	3523.2	1.31
2 CPU 4U	fcgi_inet	2517.6	0.95
2 CPU 4U	base_uds	2643.9	1.00
2 CPU 4U	sharedmem	3027.1	1.14
2 CPU 4U	passfd	3398.4	1.29
4 CPU 4U	fcgi_inet	3441.1	0.98
4 CPU 4U	base_uds	3525.1	1.00
4 CPU 4U	sharedmem	3980.1	1.13
4 CPU 4U	passfd	4363.6	1.24

Table 5.2: Peak Throughput

The performance of FastCGI with Internet sockets is worse than FastCGI with Unix domain sockets over all the configurations. The difference is greatest in the

Architecture	Average Performance Relative to Base Case
fcgi_inet	0.93
base_uds	1.00
sharedmem	1.14
passfd	1.27

Table 5.3: Average Performance Relative to Base Case Over All Server Configurations

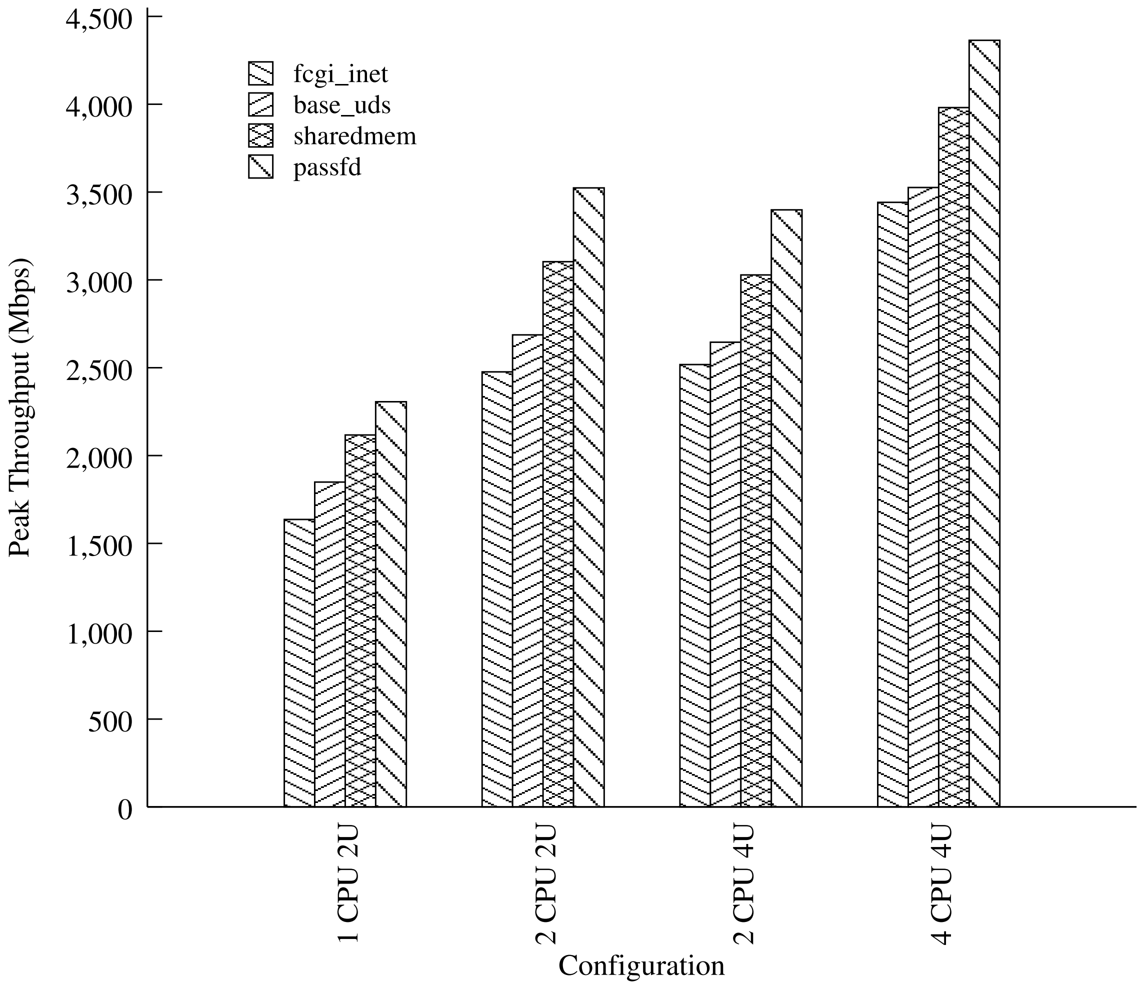


Figure 5.2: Peak Throughput

configuration with one processor, as the peak throughput of FastCGI with Internet sockets is 0.89 times that of the base case (Table 5.2, Configuration 1 CPU 2U). Therefore, FastCGI with Unix domain sockets is preferred when web servers and application servers are co-located on the same machine. Compared to FastCGI with Internet sockets, FastCGI with Unix domain sockets has better performance because there is no TCP processing overhead on data sent between the web server and the application server. On the other hand, the performance difference decreases as the number of CPU increases. We observe that a lower percentage of CPU time is spent in TCP processing as the number of CPU increases, and we believe this is because TCP processing is shared by the additional processors.

Table 5.3 shows the average performance relative to the base case over all server configurations. The first column of Table 5.3 lists the FastCGI architecture and the second column shows the average performance of the respective FastCGI architecture relative to the base case over all server configurations. We can see that FastCGI with shared memory has on average 1.14 times the throughput of the base case over all configurations. FastCGI with PASSFD has on average 1.27 times the throughput of the base case. In the server configuration with 4 CPUs (4 CPU 4U), its peak throughput reaches 4363.6 Mbps.

Figure 5.2 shows a graphical view of peak throughput of our experiments. The y-axis shows peak throughput in Mbps, and the x-axis shows the different server architectures clustered by server configuration. For example, the first cluster shows peak throughput of the four FastCGI architectures under the 1 CPU 2U configuration. This figure illustrates that FastCGI with shared memory and FastCGI with PASSFD have significant performance improvements over the FastCGI with Unix domain sockets base case.

These experiments show the significant performance gains that can be obtained by using FastCGI with shared memory and FastCGI with PASSFD when the web server and the application server are co-located on the same machine. As shown in the next section, this is due to the reduction in copying overheads. However, what is surprising and interesting is that FastCGI with PASSFD produces significantly greater peak throughput than FastCGI with shared memory. In the next section, we measure copying costs and compare the different architectures in order to better understand the performance differences.

5.2 Copying Cost Comparison

The previous section uses a black-box approach to look at the peak throughput performance improvement of the two FastCGI enhancements. Since our goal is to reduce the amount of data copied in the FastCGI architecture during response delivery, we need to take a white-box approach to understand the copying costs. These further insights direct us to focus on specific system dynamics in later sections.

As previously mentioned, response delivery in FastCGI architectures utilizes socket communication. During a socket *send()* or *recv()* system call, the *__copy_user()* routine copies data across the user and kernel boundary (from and to user space). As this operation is performed by the processor, a higher percentage of CPU time spent in the routine implies higher copying cost. Therefore, we compare the cost of data copying by observing the percentage of CPU time spent in the *__copy_user()* routine in each of the experiments.

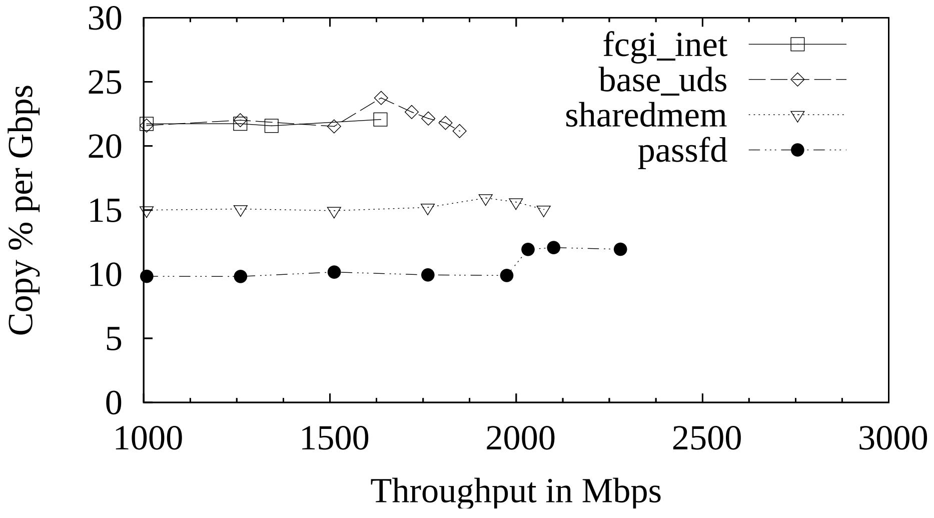
Because FastCGI with shared memory and FastCGI with PASSFD both aim to reduce copying redundancy, we anticipate the percentage of CPU time spent in the *__copy_user()* routine to be lower than in FastCGI with Unix domain sockets.

Although we expect lower copying costs in both the shared memory and PASSFD optimizations, we also consider the efficiency of these implementations as server utilization increases. We therefore examine the ratio of the percentage of CPU time spent in the *__copy_user()* routine across all processors over the data throughput (called copying cost), resulting in a measure which is the percentage of CPU time spent copying data per Gbps of data throughput (%/Gbps). In multiprocessor setups, the percentage of CPU time spent in the *__copy_user()* routine is the average of percentage of CPU times each processor spent in the *__copy_user()* routine.

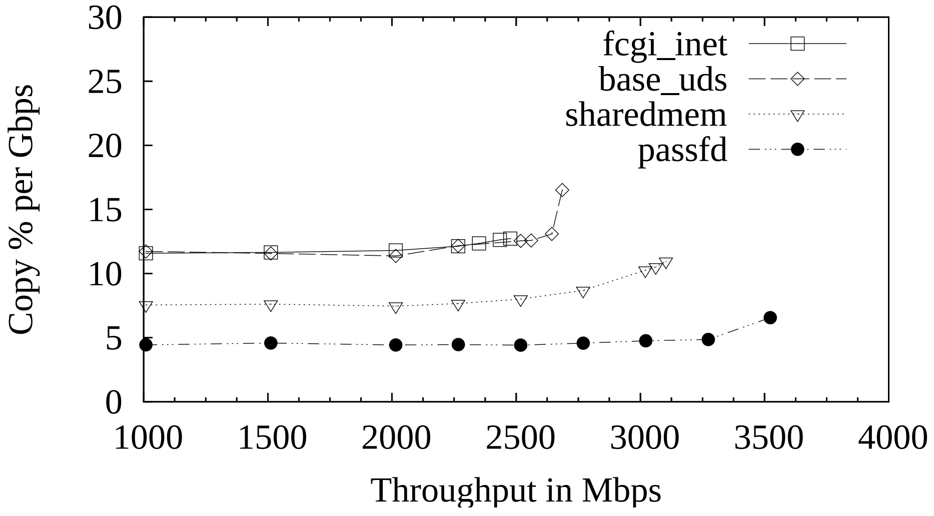
5.2.1 Copying Costs at Fixed Server Throughputs

In Section 4.1.2, we explained there are no client timeouts so no overload conditions can occur. Hence, our performance analysis focuses on conditions before saturation. To compare memory copying costs among architectures, we examine their respective copying costs at comparable throughput rates.

Figures 5.3 and 5.4 show that the copying costs of FastCGI with Internet sockets are very similar to copying costs of FastCGI with Unix domain sockets. This demonstrates that the performance difference we see in Section 5.1 between these two architectures does not result from copying costs differences. In the uniprocessor configuration (Figure 5.3a) at 1008 Mbps, FastCGI with Internet sockets and FastCGI with Unix domain sockets have very similar copying overheads which is around 21.6 %/Gbps. FastCGI with shared memory has lower copying costs of 15 %/Gbps. FastCGI with PASSFD has the lowest copying costs of 10 %/Gbps. The



(a) Copying Cost 1 CPU 2U

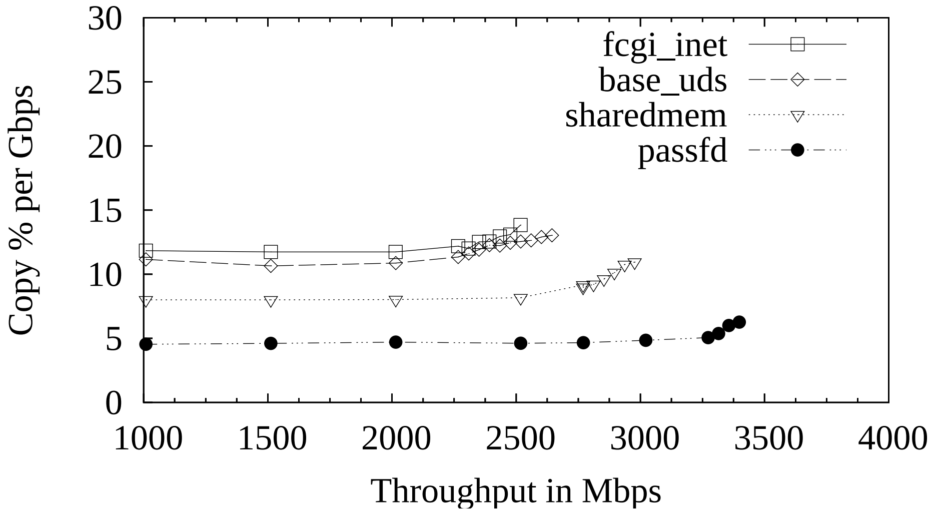


(b) Copying Cost 2 CPU 2U

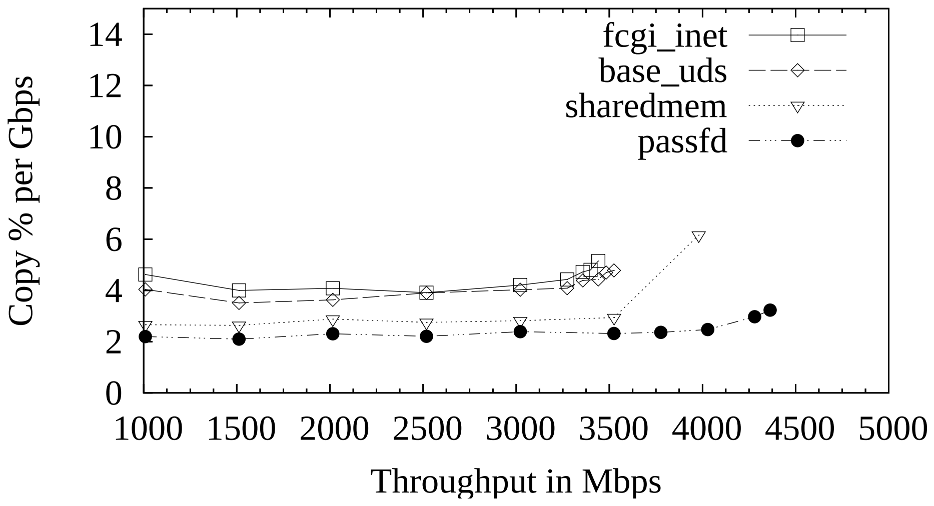
Figure 5.3: Copying Costs in Configurations 1 CPU 2U and 2 CPU 2U

same pattern holds in Configurations 2 CPU 2U, 2 CPU 4U and 4 CPU 4U (Figures 5.3b, 5.4a, 5.4b) in which FastCGI with Internet sockets and FastCGI with Unix domain sockets have the highest and most similar copying costs. FastCGI with shared memory has lower copying costs and FastCGI with PASSFD has the lowest copying costs. When compared to the base case, both FastCGI with shared memory and FastCGI with PASSFD incur significantly less copying overhead.

Notice that in Figures 5.3a, 5.3b, 5.4a and 5.4b, as the number of processors increases, the differences in copying costs between the base case and both FastCGI with shared memory and FastCGI with PASSFD decrease. For example, in the



(a) Copying Cost 2 CPU 4U



(b) Copying Cost 4 CPU 4U

Figure 5.4: Copying Costs in Configurations 2 CPU 4U and 4 CPU 4U

uniprocessor configuration (Figure 5.3a) at 1008 Mbps, the difference in copying costs between the base case and FastCGI with shared memory is 6.6 %/Gbps. Then, in Configuration 2 CPU 2U (Figure 5.3b) at 1008 Mbps, the base case has a copying cost of 11.6 %/Gbps and FastCGI with shared memory has a copying cost of 7.6 %/Gbps. The difference in copying costs between the base case and FastCGI with shared memory decreases to 4 %/Gbps. The differences in copying costs between the base case and FastCGI with shared memory decrease further in Configuration 4 CPU 4U. Similarly, the differences in copying costs between the base case and FastCGI with PASSFD decrease as the number of processors increases.

Although the differences in copying costs between the base case and both FastCGI with shared memory and FastCGI with PASSFD decrease as the number of processors increases, the ratios of the copying costs remains the same as the number of processors increases. We examine the ratios of the copying cost of FastCGI with shared memory over the copying cost of the base case at comparable throughput rates (at 1008 Mbps, 1260 Mbps and 1511 Mbps) before copying costs increase near saturation. In Configuration 1 CPU 2U (Figure 5.3a) at 1008 Mbps, FastCGI with shared memory has a copying cost of 15.0 %/Gbps and FastCGI with Unix domain sockets has a copying cost of 21.6 %/Gbps. The ratio of these copying costs (15.0 / 21.6) is 0.69. The ratios remain similar at 1260 and 1511 Mbps in Configuration 1 CPU 2U. In Configuration 2 CPU 2U (Figure 5.3b) at 1008 Mbps, FastCGI with shared memory has a copying cost of 7.6 %/Gbps and the base case has a copying cost of 11.6 %/Gbps. The ratio (7.6 / 11.6) is 0.66. The ratios remain similar at 1260 and 1511 Mbps in Configuration 2 CPU 2U.

In the four configurations, the copying costs of FastCGI with shared memory are around 0.7 times those of the base case. We also examine the same ratios of the copying costs of FastCGI with PASSFD over those of the base case, and they are around 0.4 in all configurations. Since the ratios of copying costs remain consistent as the number of processors increases, the relative performance improvements of FastCGI with shared memory and FastCGI with PASSFD remain the same as the number of processors increases.

In Configurations 2 CPU 2U, 2 CPU 4U and 4 CPU 4U, we notice that the copying costs increase as the server approaches saturation. For example, in Configuration 2 CPU 2U (Figure 5.3b), FastCGI with shared memory has a copying cost of 7.6 %/Gbps at the throughput rate of 2015 Mbps. The copying costs slowly increase as the throughput increases beyond 2015 Mbps to around 10.3 %/Gbps at around 3000 Mbps. Other curves in Configuration 2 CPU 2U also follow a similar pattern. Copying costs in Configurations 2 CPU 4U and 4 CPU 4U (Figures 5.4a and 5.4b) also have a gradual increase near saturation. However, we do not see this pattern in Configuration 1 CPU 2U. In Figure 5.3a, FastCGI with Unix domain sockets has a copying cost of 21.5 %/Gbps at 1511 Mbps. At 1638 Mbps, its copying cost increases to 23.8 %/Gbps, but beyond 1638 Mbps the copying costs gradually decrease. Copying costs of FastCGI with shared memory and those of FastCGI with PASSFD also show an increase, followed by a gradual decrease. From our *OProfile* statistics, we realize that the percentage of CPU time spent in network processing functions increases significantly near saturation. Therefore, copying costs in Configuration 1 CPU 2U do not increase as the server approaches saturation.

On top of the above copying costs comparisons, there are two additional observations that lead to further questions. In Chapter 2 and Chapter 3, we explained that both FastCGI with shared memory and FastCGI with PASSFD reduce the amount of data copied by two-thirds. Initially, we expected the two techniques to have similar performance. However, Section 5.1 shows that FastCGI with PASSFD has a significantly higher peak throughput than FastCGI with shared memory. We also notice that FastCGI with PASSFD has a significantly lower copying costs compared to copying costs of FastCGI with shared memory. Although both techniques show significant performance improvements when compared to FastCGI with Unix domain sockets, we are interested in identifying the additional factors that differentiate their performance. Secondly, we seek to understand factors that cause copying costs to increase as the server approaches saturation. In Section 5.3, we perform further analysis to determine the answers to the above questions.

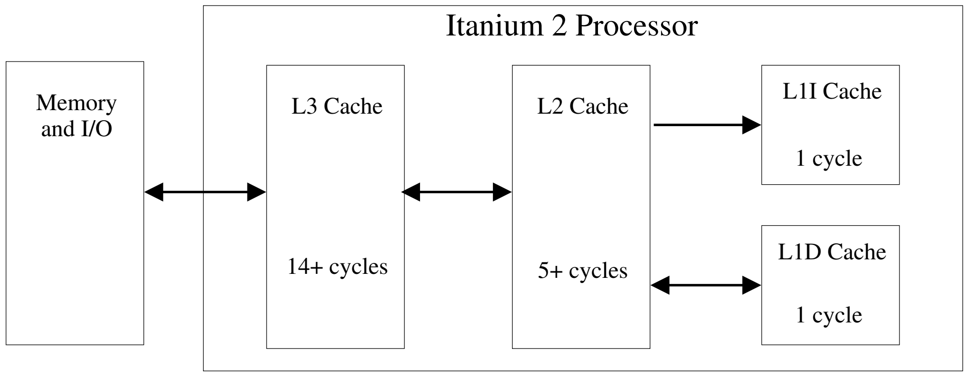
5.3 Cache Analysis

In Section 5.2, we identify two questions that require further understanding of system behaviour to explain. First, even though both FastCGI with shared memory and FastCGI with PASSFD reduce the amount of data copied by two-thirds, we need to understand why FastCGI with PASSFD has significantly higher peak throughputs and lower copying costs. Second, it is important to understand factors causing copying costs to increase near server saturation in the multiprocessor configurations.

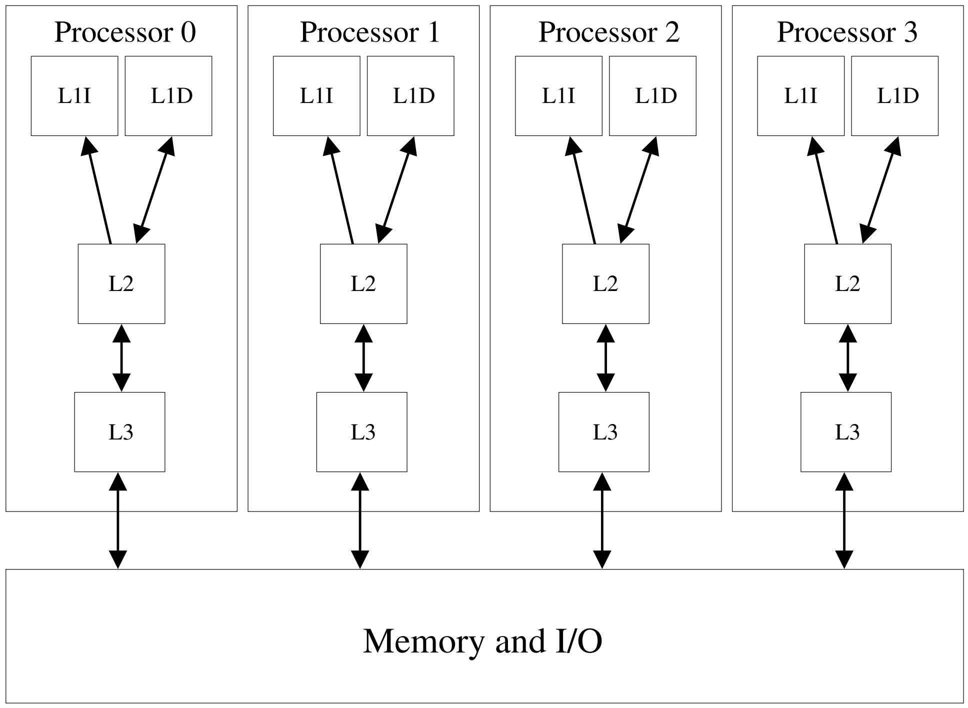
In this section, we examine the processor cache hierarchy, which aims to improve data access speed by exploiting spacial and temporal locality characteristics in program execution.

The Itanium 2 processor memory subsystems (Figure 5.5a) uses a three-level cache hierarchy. Our version of the Itanium 2 processor contains one core per processor. Each core consists of a first-level instruction cache (16 KB), a separate first-level data cache (16 KB), unified second-level cache (256 KB) and unified third-level cache (6 MB). Figure 5.5b shows the processor setup of our server. Each of the processors has its own three-level cache hierarchy but accesses the same memory.

We focus our analysis on data cache misses as we expect the shared code segments are unlikely to be subjected to cache pressure. In Section 5.1.2, FastCGI with shared memory and FastCGI with PASSFD both show consistent performance improvements across all four configurations when compared to the base case. Then, in Section 5.2, FastCGI with shared memory and FastCGI with PASSFD have consistent copying costs relative to the base case across all configurations. Therefore, we only analyze results captured in Configuration 2 CPU 2U.



(a) Itanium 2 Processor Cache Hierarchy



(b) Processor Setup in Our Server

Figure 5.5: Itanium 2 Memory Subsystem

We use *OProfile* [17, 10] to capture the three cache miss events. They are `L1D_READ_MISSES`, `L2_MISSES` and `L3_MISSES`. `L1D_READ_MISSES` is incremented when a data read does not reside in first-level data cache. `L2_MISSES` and `L3_MISSES` are incremented when there is a read or write cache miss in the respective cache. Since L2 and L3 are both unified caches, there are no separate events to distinguish data misses from instruction fetch misses. Our results show there are very few first level instruction cache misses, so we believe that most L2 and L3 misses are data fetch misses. Profiling cache misses allows us to understand cache behaviour of the different FastCGI architectures at different data rates.

5.3.1 Shared Memory vs PASSFD

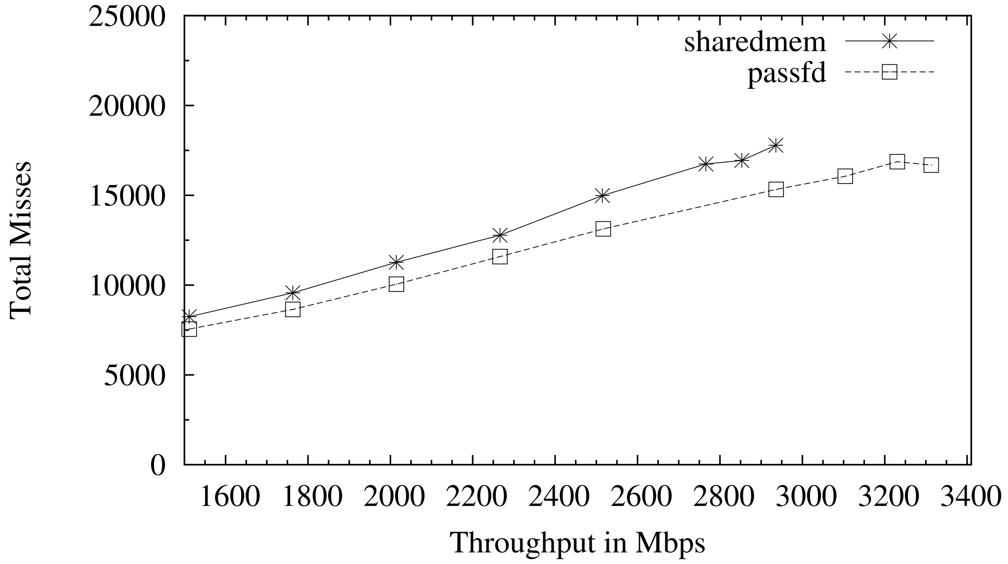
Because both FastCGI with shared memory and FastCGI with PASSFD require only one data copy between user and kernel space during response delivery, we expect them to have similar throughput and copying costs. However, as seen in Section 5.1 and 5.2 FastCGI with PASSFD has significantly higher peak throughput (Figure 5.2) as well as lower copying costs (Figure 5.3 and 5.4).

In Figures 5.6 and 5.7, we plot the data rates on the x-axis and the number of cache misses on the y-axis, and we examine the number of cache misses at each level as the server throughput increases. In all L1 Data, L2 and L3 caches and for all data rates, FastCGI with shared memory incurs a higher number of cache misses than FastCGI with PASSFD.

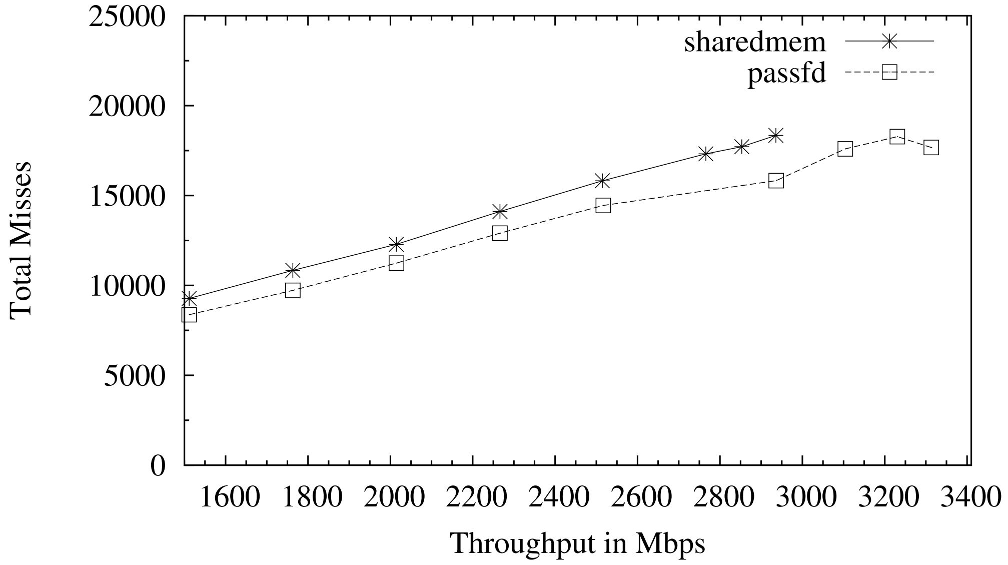
Let us revisit the difference between the two optimizations. In FastCGI with shared memory, the application server (let us call it APP 1) puts the response into shared memory and then notifies the web server that the response is ready. Since APP 1 has completed its task, the operating system switches to another process that is ready to run. However, this process may be another application server (let us call it APP 2). When APP 2 runs, it reuses the cache and may invalidate APP 1's response. Later, when the web server sends APP 1's response, the response may not be in the cache.

On the other hand, FastCGI with PASSFD delivers the response directly from the application server before notifying the web server. Because there is usually no process switch, the response is likely to be in the cache when it is sent to the client, resulting in fewer cache misses.

A higher number of cache misses translates to more CPU cycles spent per data access operation. Consequently, more CPU cycles spent at a constant data rate results in a higher copying cost. Therefore, the copying costs and cache misses are correlated and that the fewer cache misses for FastCGI with PASSFD are what provide it with significantly better performance than FastCGI with shared memory.



(a) L1 Data Read Misses

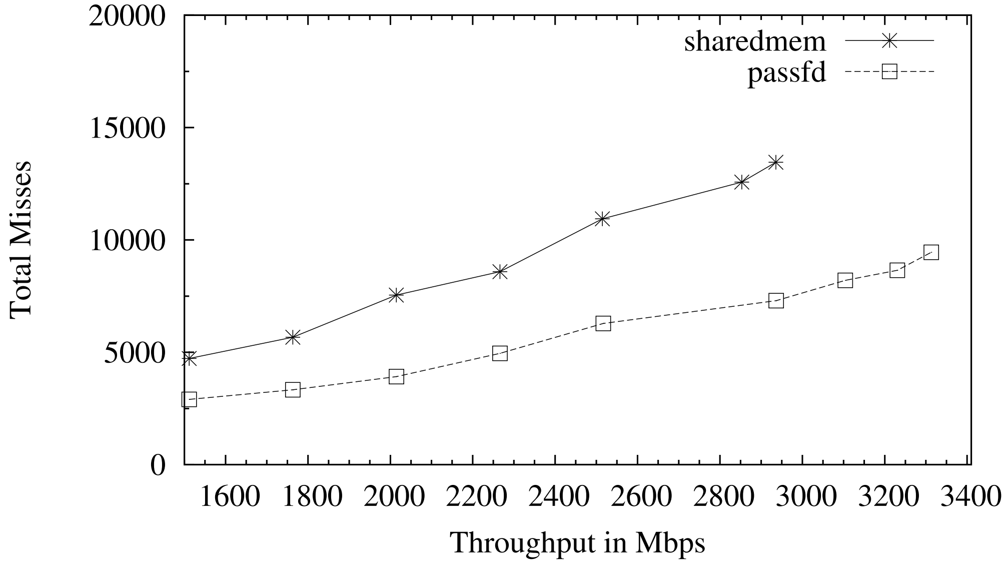


(b) L2 Misses

Figure 5.6: Cache Misses in Configuration 2 CPU 2U, L1 and L2 Caches

5.3.2 Copying Costs Increase Near Saturation

In Section 5.2, we notice an increase in copying cost as the server reaches saturation. Since our experiments use a closed system to avoid wasting CPU resources from early client termination, we cannot analyze copying cost after server saturation. However, if we extrapolate the copying costs trend, we would notice copying costs increase very rapidly near saturation and in saturation. Therefore, we look at pro-



(a) L3 Misses

Figure 5.7: Cache Misses in Configuration 2 CPU 2U, L3 Cache

cessor cache statistics to formulate a potential cause for such phenomena, which is useful for future research in designing more efficient response delivery mechanisms. Since we observe a similar increase in copying costs in Configurations 2 CPU 2U, 2 CPU 4U and 4 CPU 4U, we only analyze results captured in Configuration 2 CPU 2U.

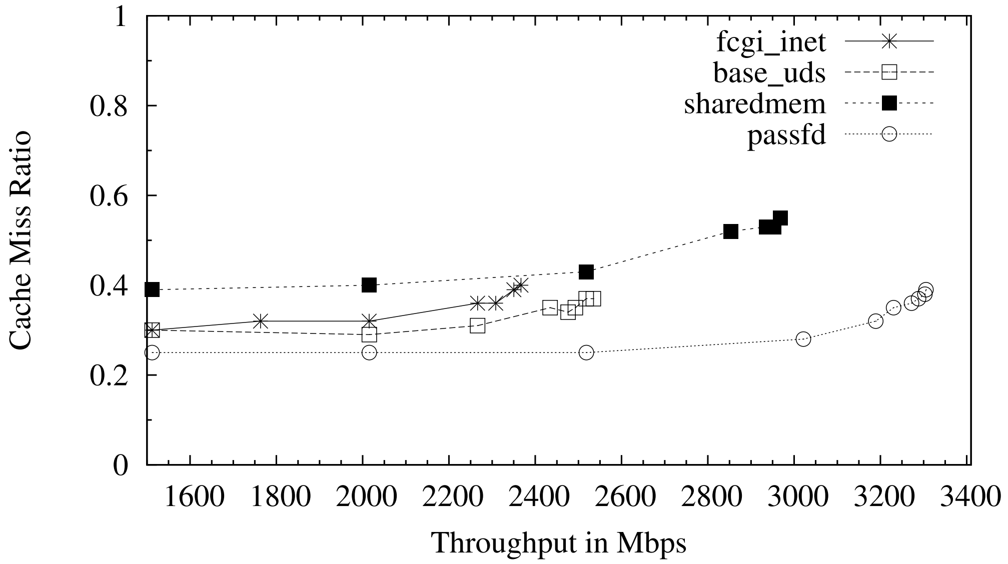


Figure 5.8: Cache Miss Ratio in Configuration 2 CPU 2U, L3 Cache

We examine the cache miss ratio in the *__copy_user()* function across all processors. It is measured as the total number of cache misses in *__copy_user()* divided by the total number of cache references in *__copy_user()*. Since our version of *OProfile* is only able to capture the number of references to the L3 cache (with the `L3_REFERENCES` event), we examine the cache miss ratio of the L3 cache. Figure 5.8 plots the data rates on the x-axis and the L3 cache miss ratio on the y-axis.

FastCGI with PASSFD has a flat cache miss ratio curve in the beginning, but it increases slightly as the server approaches saturation, which is very similar to its copying costs trend. Cache miss ratio curves of the other architectures also follow a similar pattern. They all have a fairly flat beginning and increase slightly as the server approaches saturation.

Notice that the cache miss ratio curve of FastCGI with Unix domain sockets is lower than the cache miss ratio curve of FastCGI with shared memory. This does not mean FastCGI with shared memory has lower performance, because in Section 5.2 we showed that FastCGI with Unix domain socket has higher copying costs than FastCGI with shared memory in all server configurations.

In addition, we observe from our server logs that the average number of busy application servers (i.e., the average number of application servers servicing a dynamic request) follows a similar trend as the cache miss ratio curves in Figure 5.8. For example, at lower data rates, there are on average 16 application servers handling requests at the same time. The average number of busy application servers increases to 160 as the server is close to saturation. The IA-64 Linux kernel uses an address space number as part of the search key in the translation lookaside buffer (TLB) lookup, so processor caches do not need to be flushed during process switches [24]; therefore, the higher cache miss ratios near server saturation are not caused by the flushing of processor caches. We believe the higher cache miss ratios are caused by a larger working set size from a higher number of busy application servers near server saturation. The increase in the cache miss ratio (Figure 5.8) implies a higher number of cache misses at high data rates, which increases the frequency of memory access, resulting in an increase in copying costs near server saturation.

5.4 Performance with a Representative Workload

Suzumura *et al.* [39] analyzed the average data size in SPECweb2005 [5]. For the banking scenario, the average response size is 34.8 KB. For the e-commerce scenario and the support scenario, the average response sizes are 143.9 KB and

78.5 KB respectively. They showed that in the e-commerce scenario, their base server spent 15.7% of the total CPU time on memory copying. This represents a significant opportunity to improve performance by reducing the amount of data copied.

They then showed their zero-copy static content optimization reduces CPU usage of memory copying from 15.6% to 3.4% and from 12.4% to 4.4% in the e-commerce and the support scenario. This translates to a 22.2% and 44.4% peak throughput improvement over their base server. However, there is not much improvement in the banking scenario.

In Section 5.1.1, we see that the performance of FastCGI with PASSFD is on average 1.29 times the performance of FastCGI with Unix domain sockets under workloads with different response sizes. In Section 5.1.2, we see that FastCGI with PASSFD also shows a performance improvement of 1.27 times on average when compared to FastCGI with Unix domain sockets under different server configurations. From Section 5.2, the copying costs of FastCGI with PASSFD are 0.4 times that of FastCGI with Unix domain sockets. Similar to Suzumura's technique, FastCGI with PASSFD shows significant performance improvements across different response sizes. It also shows reductions in CPU usage of memory copying across both uniprocessor and multiprocessor environments.

In Section 5.1.1, we explained that FastCGI with PASSFD provides little performance improvement with a response size of 32 KB, which is similar to Suzumura's results in the banking scenario. With a response size of 64 KB, FastCGI with PASSFD has a peak throughput of 1.15 times that of the base case. Although it is lower than the average improvement, we believe it will provide performance improvement in the SPECweb2009 support scenario, which has a similar average response size.

We conjecture that FastCGI with PASSFD will provide good performance improvements in the e-commerce and support scenario in SPECweb2005 and SPECweb2009. Furthermore, FastCGI with PASSFD reduces data copying for dynamically generated data, which is something that Suzumura's zero-copy optimization does not do. This reduction can translate to further performance improvement in workloads with more dynamic content.

5.5 Summary

In this chapter, we have compared the peak performance of each FastCGI architecture under workloads with different response sizes and under different server configurations. FastCGI with PASSFD obtains peak throughputs of 1.26 to 1.31 times that obtained with FastCGI with Unix domain sockets under workloads with different response sizes above 128 KB. FastCGI with PASSFD also performs the best under different server configurations, obtaining an average peak throughput of 1.27 times that obtain with FastCGI with Unix domain sockets.

We examine the percentage of CPU time consumed in copying data between user and kernel space per unit of data throughput. The copying costs of FastCGI with PASSFD are reduced to 0.4 times that of the base case, and the copying costs of FastCGI with shared memory are 0.7 times that of the base base. This is particularly interesting because these two approaches require the same number of data copies. We also notice a significant increase in copying costs as the server approaches saturation.

Using hardware performance counters to understand cache performance, we show that FastCGI with PASSFD has a lower number of cache misses than FastCGI with shared memory. The number of cache misses is significantly lower for FastCGI with PASSFD because the response is sent to the client by the application server, so the response is likely to remain in the cache when the response is sent, resulting a lower CPU usage.

In the cache miss ratio analysis, we show that cache miss ratios increase near server saturation, similar to the way copying costs increase near server saturation. We believe that the increase in the average number of busy application servers near server saturation causes the cache miss ratio to increase, which increases copying costs near server saturation.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we explore the opportunity to improve the performance of web servers that handle dynamic content when the web server and the application server are co-located on the same machine. We examine the benefits of using Unix domain sockets and of reducing the amount of data copied during response delivery. We propose and implement a technique that passes the client socket descriptor from the web server to the application server (called FastCGI with PASSFD), permitting the application server to reply directly to the client, thereby reducing the amount of data copied by two-thirds. This technique requires modifications only to the web server and the FastCGI library, making it cost-effective, easy to implement, and easy to deploy in a variety of operating systems, web servers and application servers.

We compare the performance of FastCGI with PASSFD and the performance of FastCGI with shared memory against FastCGI with Unix domain sockets under both uniprocessor and multiprocessor environments. The shared memory approach has an average peak throughput of 1.14 times that of FastCGI with Unix domain sockets. FastCGI with PASSFD has the best average peak throughput of 1.27 times that of FastCGI with Unix domain sockets across all four server configurations. Copying costs of FastCGI with PASSFD are reduced to 0.4 times that of FastCGI with Unix domain sockets, and FastCGI with shared memory has copying costs of 0.7 times that of FastCGI with Unix domain sockets.

Interestingly, we find that although FastCGI with PASSFD and FastCGI with shared memory require the same amount of data copying, FastCGI with PASSFD has a better average peak throughput and lower copying costs. With further investigation, we find that FastCGI with shared memory has a significantly higher number of cache misses than FastCGI with PASSFD. Since FastCGI with PASSFD sends the response directly to the client from the application server, it has a higher time locality, and the response is more likely to be in the cache than when the response is sent from the web server (as in FastCGI with shared memory). A higher number of cache misses translates to more CPU cycles spent per data access operation, which in turn results in higher copying costs.

We also examine the L3 cache miss ratio of all architectures and notice an increase in the L3 cache miss ratio similar to the increase in copying costs near server saturation. We identify a similar increase in the average number of busy application servers near server saturation, which leads us to believe that cache contention caused by the higher number of busy application servers causes higher cache miss ratios, which increases copying costs when the server is close to saturation.

6.2 Future Work

6.2.1 Zero-Copy FastCGI with PASSFD

In the future, we plan to further reduce the amount of data copied by utilizing *sendfile()* with our PASSFD technique. FastCGI with PASSFD sends the response from the FastCGI library, reducing the amount of data copied to one-third and the copying costs to 0.4 times that of FastCGI with Unix domain sockets. The only remaining redundant data copy occurs when the FastCGI library sends the response to the client. The data is copied from the user space of the FastCGI library to the kernel space before it is sent by the network device. It may be possible to avoid this data copy by using the *mmap()* and *sendfile()* system calls. First, the FastCGI library would use *mmap()* to map its content buffer to a temporary file. Then, since the temporary file would be cached in the kernel memory, we could use the *sendfile()* system call on the temporary file to send the contents directly to the network device, bypassing the data copy from the user space to the kernel space [37]. In Linux there is currently no way to determine when the data has been sent and therefore when the buffer can can be reused. However, FreeBSD's *sendfile()* may provide support for such an operation.

6.2.2 Automatic System Tuning

With increasing accessibility in the last few years, the Internet has evolved from an information source to a participation platform. Social networks and multimedia services like Facebook [7] and YouTube [45] have become very popular. With different types of media content and different types of Internet applications, workloads have become increasingly heterogeneous and variable.

Until now, system tuning has been done manually by system administrators following tuning principles and accumulated experience. These tunings tend to be static and time consuming and can sometimes under perform as the workload changes. One example of a workload change is called the Slashdot effect, when a massive increase in traffic to a website is caused by a link from a popular site. These workload changes often cause the server to under perform and can sometimes cause websites or services to become inaccessible.

By exploring different system parameters, such as the number of web servers, the number of application servers, the number of available connections and how requests are queued for application servers to handle, we can formulate auto tuning agents using statistical methods or artificial intelligence techniques to perform automated system tuning. This research direction would be valuable as Internet applications continue to evolve and become more complex.

6.2.3 Reducing Data Copying with Splice and Tee

Three new system calls have been introduced in version 2.6.17 of the Linux kernel. They allow an application to move data between file descriptors without copying the data between user and kernel space by utilizing a pipe as an in-kernel data buffer. The *splice()* system call [20] moves data between a file descriptor and a pipe, the *tee()* system call [20] copies data between two pipes and the *vmsplice()* system call [22] maps user memory segments into a pipe's data buffer. It would be interesting to explore how these calls might be used to reduce copying and improve performance

Bibliography

[1] Gaurav Banga and Peter Druschel. Measuring the Capacity of a Web Server Under Realistic Loads. *World Wide Web*, 2(1-2):69–83, 1999. 31

[2] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel*, chapter 16. O'Reilly Media, Inc., 2006. 16

[3] Tim Brecht, David Pariag, and Louay Gammo. accept()able Strategies for Improving Web Server Performance. In *ATEC '04: Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, 2004. 32

[4] Standard Performance Evaluation Corporation. SPECweb99 Design Document, 1999. Available at `http://www.spec.org/web99/docs/whitepaper.html`. 31, 32

[5] Standard Performance Evaluation Corporation. SPECweb2005, 2005. Available at `http://www.spec.org/web2005/`. 9, 55

[6] Standard Performance Evaluation Corporation. SPECweb2009, 2009. Available at `http://www.spec.org/web2009/`. 31

[7] Facebook, Inc. Facebook. `http://www.facebook.com`. 60

[8] Roy T. Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Tim Berners-Lee, Larry Masinter, and Paul Leach. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999. 27

[9] Roy T. Fielding and Gail Kaiser. The Apache HTTP Server Project. *Internet Computing, IEEE*, 1(4):88 –90, July/August 1997. 7, 23

[10] Tammy Fox. *Red Hat Enterprise Linux 5 Administration Unleashed*, chapter 22. Sams, 2007. 37, 51

[11] Tim Francis, Eric Herness, Rob High, Jim Knutson, Kim Rochat, and Chris Vignola. *Professional IBM WebSphere 5.0 Application Server*. Wiley & Sons, Inc., New York, 2002. 7

[12] Paul Heinlein. FastCGI. *Linux Journal*, 55, November 1998. 7

[13] Thomas Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems*. Charles River Media, Inc., 2004. 27

[14] IBM Corp. IBM HTTP Server. `http://www-01.ibm.com/software/webservers/httpservers/`. 7

[15] David Iseminger. *Microsoft Network Services Developer's Reference Library*, chapter 6. Microsoft Press, 2000. 28

[16] iSpeech, Inc. iSpeech. `http://www.ispeech.org`. 31

[17] John Levon and Philippe Elle. OProfile - A System Profiler for Linux. Available at `http://oprofile.sourceforge.net`. 37, 51

[18] Anthony Jones and Jim Ohlund. *Network Programming for Microsoft Windows*. Microsoft Press, 1999. 28

[19] Philippe Joubert, Robert B. King, Richard Neves, Mark Russinovich, and John M. Tracey. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *ATEC '01: Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, 2001. 9

[20] Linus Torvalds. Explaining splice() and tee(). `http://kerneltrap.org/node/6505`. 60

[21] Linux Man Page. TCP. `http://linux.die.net/man/7/tcp`. 27

[22] LWN.net. The splice() weekly news. `http://lwn.net/Articles/181169/`. 60

[23] Martin Blom. ESXX – Friendly Server-side JavaScript. `http://esxx.org/`. 7

[24] David Mosberger and Stephane Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall PTR, 2002. 55

[25] David Mosberger and Tai Jin. httperf: A Tool for Measuring Web Server Performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998. 32

[26] Open Market, Inc. FastCGI: A High-Performance Web Server Interface, April 1996. Available at `http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm`. 7

[27] Linux Man Page. sendfile. `http://linux.die.net/man/2/sendfile`. 10

[28] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, and Amol Shukla. Comparing the Performance of Web Server Architectures. In *EuroSys '07: Proceedings of the 2nd ACM European Conference on Computer Systems*. ACM, 2007. 32

[29] KyoungSoo Park and Vivek S. Pai. Connection Conditioning: Architecture-Independent Support for Simple, Robust Servers. In *NSDI '06: Proceedings of the 3rd Symposium on Networked Systems Design & Implementation*. USENIX Association, 2006. 22

[30] The FreeBSD Project. *TCP - FreeBSD Kernel Interfaces Manual*. 27

[31] Hiran Ramankutty. Inter-Process Communication - Part II. *Linux Gazette*, 105, August 2004. 16

[32] Red Hat, Inc. *TUX 2.2 Reference Manual*, 2002. 9

[33] David Robinson and Ken Coar. The Common Gateway Interface (CGI) Version 1.1. RFC 3875 (Informational), October 2004. 6

[34] Yaoping Ruan and Vivek S. Pai. Making the "Box" Transparent: System Call Performance as a First-Class Result. In *ATEC '04: Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, 2004. 22

[35] Yaoping Ruan and Vivek S. Pai. Understanding and Addressing Blocking-Induced Network Server Latency. In *ATEC '06: Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, 2006. 22

[36] Amol Shukla, Lily Li, Anand Subramanian, Paul A. S. Ward, and Tim Brecht. Evaluating the Performance of User-Space and Kernel-Space Web Servers. In *CASCON '04: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 189–201. IBM Press, 2004. 9

[37] Dragan Stancevic. Zero Copy 1: User-mode perspective. *Linux Journal*, 105, January 2003. 59

[38] Sun Microsystems, Inc. *Solaris 10 Software Developer Collection - STREAMS Programming Guide*, chapter 6. 2005. 28

[39] Toyotaro Suzumura, Michiaki Tatsubori, Scott Trent, Akihiko Tozawa, and Tamiya Onodera. Highly Scalable Web Applications with Zero-Copy Data Transfer. In *WWW '09: Proceedings of the 18th International Conference on World Wide Web*, pages 921–930. ACM, 2009. ix, 1, 9, 10, 11, 17, 39, 55

[40] Ajay Tirumala and John Ferguson. Iperf. Available at `http://sourceforge.net/projects/iperf`. 33

[41] Jeff Tranter. Exploring The sendfile System Call. *Linux Gazette*, 91, June 2003. 10

[42] Vivek S. Pai. The Flash Web Server. `http://www.cs.princeton.edu/~vivek/flash/`. 23

[43] Mike Volodarsky, Olga Londer, Brett Hill, Bernard Cheah, Steve Schofield, Carlos Aguilar Mares, Kurt Meyer, and the Microsoft IIS Team. *Internet Information Services (IIS) 7.0 Resource Kit.* Microsoft Press, 2008. 7

[44] Dan Woods, Larne Pekowsky, and Tom Snee. *The Developer's Guide to the Java Web Server: Building Effective and Scalable Server-Side Applications.* Addison-Wesley Pub, 1999. 7

[45] YouTube, LLC. YouTube. `http://www.youtube.com`. 60