Highly Scalable Web Applications with Zero-Copy Data Transfer

Toyotaro Suzumura, Michiaki Tatsubori
Scott Trent, Akihiko Tozawa, and Tamiya Onodera
Tokyo Research Laboratory, IBM Research
1623-14 Shimo-tsuruma, Yamato, Kanagawa, Japan
{toyo,mich,trent,atozawa,tonodera}@jp.ibm.com

ABSTRACT
The performance of server-side applications is becoming increasingly important as more applications exploit the Web application model. Extensive work has been done to improve the performance of individual software components such as Web servers and programming language runtimes. This paper describes a novel approach to boost Web application performance by improving inter-process communication between a programming language runtime and Web server runtime. The approach reduces redundant processing for memory copying and the context switch overhead between user space and kernel space by exploiting the zero-copy data transfer methodology, such as the *sendfile* system call. In order to transparently utilize this optimization feature with existing Web applications, we propose enhancements of the PHP runtime, FastCGI protocol, and Web server. Our proposed approach achieves a 126% performance improvement with micro-benchmarks and a 44% performance improvement for a standard Web benchmark, SPECweb2005.

Categories and Subject Descriptors
I.7.2 [**Computing Methodologies**]:
Document and Text Processing—*scripting languages*;
D.3.4 [**Software**]: Processors—*runtime environments*

General Terms
Performance, Experimentation

Keywords
Web Server, Zero Copy, FastCGI, sendfile, PHP, Scripting

1. INTRODUCTION
In recent years, software applications are increasingly being developed to adopt the Web application model via HTTP protocol in the Web 2.0 era. This rapidly growing use is dramatically increasing the performance requirements for Web application servers, and much research is being done in this area [1][2][7][10][11][33]. Another recent technological trend is for Web applications to be developed with dynamic scripting languages such as PHP, Ruby, and Python, since this approach supports agile software development environments with rich sets of library functions.

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

Such languages depend on runtime systems for high performance HTTP servers in commercial environments. The two components, the language runtime and the HTTP server are connected in two different ways. One approach is to embed the language runtime within the Web server process using the internal API provided by the Web server. The other approach is to separate the language runtime from the Web server with the use of a well-defined message exchange protocol, such as FastCGI [5]. Currently, the protocol-based communication method is becoming more popular, and the combination of Lighttpd [8] and FastCGI [5] is known to be especially fast for serving PHP Web applications. Our prior work [23] showed that Lighttpd and FastCGI outperform Apache [20] and mod_php in all SPECweb scenarios. Table 1 based on that research [23] shows the peak performance in simultaneous sessions (clients) of two Web servers, Apache and Lighttpd, with the Zend PHP Runtime available at [9].

Table 1. A comparison of peak throughput in SPECweb2005

	Banking	Ecommerce	Support
Apache + mod_php	850 sessions	1,250 sessions	1,100 sessions
Lighttpd + FastCGI	1,250 sessions	2,000 sessions	1,350 sessions

In contrast to the API approach, protocol-based communication allows the programming language runtimes to be isolated from the Web server engines, which also improves the scalability and security. However, there is redundant processing in the Web server and the programming language runtime, especially in the communication between the two components. In this paper, we address this problem and make the following contributions:

- We propose a performance optimization approach that reduces inter-process communication overhead between a Web server and a PHP runtime by using a zero-copy data transfer methodology such as the *sendfile* system call.
- In order for existing PHP applications to take advantage of this optimization, we propose an enhancement of the PHP runtime that performs lazy file I/O operations while maintaining the language semantics without any modifications to the application programs. The runtime performs no File I/O operations when the content of a file is not required for other processing, and the transmission of a file to Web clients can is done by the Web server. We implement this feature in our PHP runtime engine.

- We propose an enhancement of the FastCGI protocol to denote the location of files to be sent by a Web server, and an enhancement of the Web server so that it can insert the file content at an arbitrary location and send it using a zero-copy system call such as *sendfile*. The protocol is designed to utilize a single header to specify the transfer of multiple files.
- We evaluate the new approach using both micro-benchmarks and a standard Web application benchmark, SPECweb2005, and show that our approach yields a 126% performance improvement on micro-benchmarks and a 44% improvement in the number of supportable SPECweb2005 client sessions with reasonable response times compared to the original implementation.

The remainder of the paper is organized as follows. First, the motivation and problem statement of this research are presented in Section 2. Section 3 describes the design and implementation of our optimization approach. We present evaluations in Section 4 including micro-benchmarks and experimental results for the standard Web application benchmark, SPECweb2005. Discussion appears in Section 5, and related work is reviewed in Section 6, followed by our conclusions and future directions in Section 7.

2. Motivation – Redundancy in Interprocess Communication

This section describes the motivation and problem statement of current Web architectures, focusing on their redundant processing characteristics.

2.1 Current Web architectures

Current Web applications generally use a three-tier architecture, consisting of an HTTP server, a programming language runtime, and database servers. The HTTP server and the programming language runtime are connected via a SAPI (Server Abstract API), of which there are two main types. One type of SAPI is the Apache [20] mod_php-like approach in which a language runtime is embedded in the same address space as the Web server process, connected via the proprietary API defined for the Apache HTTP server. The other SAPI type uses a language runtime running as a different process from the Web server, and the two processes communicate with a well-defined protocol such as FastCGI [5].

FastCGI is a variation of the earlier Common Gateway Interface (CGI) for interfacing interactive programs with an HTTP server. It avoids the "one new process per request" model of CGI , which limits efficiency and scalability. Instead of creating a new process for every request, FastCGI can use a single persistent process which handles many requests over its lifetime. The other benefit of using the FastCGI protocol is to allow the isolation of the programming language runtime and the Web server. For a language runtime developer, using FastCGI means the runtime will work with any Web server that understands the FastCGI protocol. For improved performance, FastCGI applications can even be distributed among multiple machines, supporting high scalability. The currently recommended default configuration for PHP with the Lighttpd Web server is to use FastCGI.

2.2 Drawbacks in protocol-based inter-process communication

As described in the previous subsection, protocol-based communication methods provide isolation and scalability, but redundant processing occurs in the language runtime and Web server. For example, consider a PHP script that prints out a header part, the content of a file, and then a footer part. The header part and the footer part may be dynamically generated, but the file in the middle is a static file or a cached result file retrieved earlier from a database. Many projects [15] [30] are pursuing caching technology for dynamic Web applications. Figure 1 illustrates the series of steps from the point when the Web server receives an HTTP request and sends the packet via the FastCGI protocol to the PHP runtime, the PHP runtime executes a PHP script, and finally responds by sending the generated HTML back to the HTTP server via FastCGI. The figure also shows where memory copying between the OS kernel space and the user space occurs. The first notable copying operation (C3) occurs in retrieving the content of a file that exists at the kernel level or in a disk cache, when the data is copied to the user-level buffers, the FastCGI and PHP runtime buffers. The second notable copying operation is C4. When the FastCGI communication is done with shared memory (when available), the PHP runtime buffer copies the content of the buffer to shared memory, and then the Web server retrieves the content from the shared memory and copies it again to a buffer in the Web server (C5) and the kernel buffer (C6), and if the content of the file for C3 is sufficiently large, then the memory copying costs for C3, C4, C5, and C6 are similar. These memory copying operations incur processing costs and lead to overall performance degradation of the Web application server. Table 2 shows the amounts of data for each scenario in SPECweb2005. This data is copied at C3, C4, C5, and C6. SPECweb2005 uses dynamic padding files that consist of random characters used to emulate the typical sizes of Web content, but which are stored as static files.

Table 2. Average data size for each SPECweb2005 scenario

	Banking	Ecommerce	Support
Average Data Size (KB)	34.8	143.9	78.5

A PHP runtime needs to dynamically retrieve the content of the file and transmit it to the Web server, which causes the copying operations at C3, C4, C5, and C6. According to our experiments with our PHP runtime using the *oprofile* profiling tool [29] in the SPECweb Ecommerce scenario, 15.7% of the total CPU time is spent on memory copying operations. There is good opportunity for improving Web application performance by reducing such memory copying operations.

3. Interprocess Communication Optimization

In this section we describe an optimization approach that addresses the problems mentioned in the previous section.

3.1 Overview of Our Approach

Our approach is to leverage operating system support, especially functions which use zero-copy data transfer [3][24] such as the *sendfile* system call supported by state-of-the-art operating systems to reduce redundant memory copying operations and interactions between user space and kernel space. The *sendfile* system call supports a zero-copy data path to transfer data from a file descriptor to a socket without CPU processing or context switching between kernel and user space. The data is immediately read from disk into the OS cache memory using Direct Memory Access (DMA) hardware, if possible. The data to be transferred is usually used directly from system buffers, without context

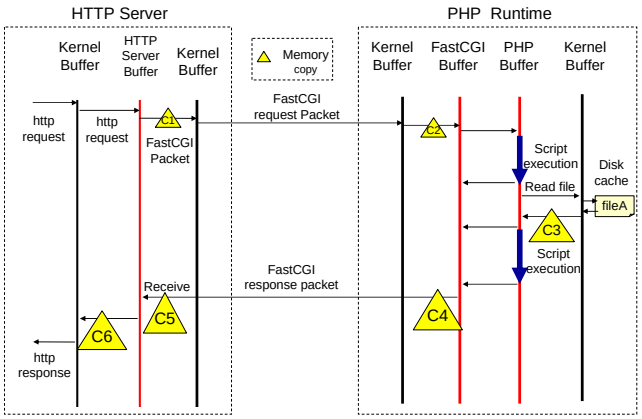


Figure 1. Interactions between HTTP server and PHP runtime.

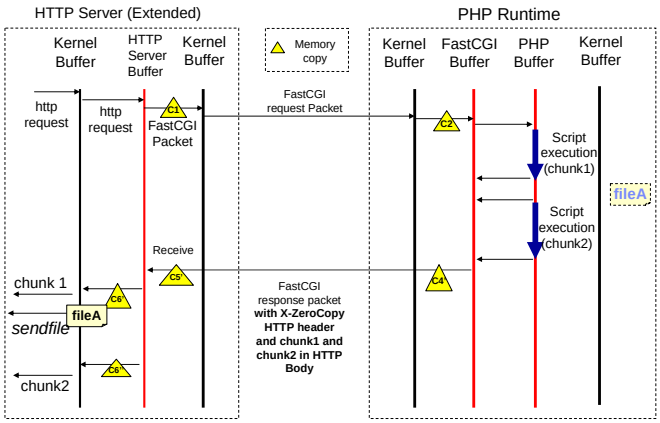


Figure 2. Interactions between HTTP server and PHP runtime with our optimization method.

switching. Thus, the usage of the *sendfile* system call significantly reduces CPU load. The *sendfile* system call is available in most modern operating systems such as Linux, AIX, Solaris, and Windows.

In addition, the *sendfile* system call is used in recent Web server implementations such as Lighttpd to send static HTML files and images files. This technique works for static files, but cannot be used when certain portions of the generated HTML are relatively static (such as loading static or cached files), but other portions are dynamic. This situation is increasingly frequent as more and more dynamically generated applications are published.

We designed and implemented an approach that can handle such mixed content while using the *sendfile* system call in a transparent way, so application developers need not modify existing applications.

3.2 Architecture Design

To transparently use the *sendfile* system call, we describe the architectural design points for interprocess communication optimization in the following sections.

3.2.1 Enhancement of PHP Runtime

To reduce the load of memory copying, we need a new type of character string object for special handling in file processing. This new character string object is called a "file-type character string object", abbreviated as FTCS object. An ordinary character string object (such as "*$a*" in *$a=file_get_contents* ("*/tmp/fileA.html*") in a PHP script) holds the entire file as a character string. In contrast, an FTCS object only holds the file name (URI) of a file. This implementation is transparent to the application program, and has the same semantics as an ordinary character string object. If the content of a file is needed for character string operations such as a regular expression operation with the *preg_match* extension, an FTCS object actually reads the file and stores its content in a form similar to an ordinary character string object. However, for information such as the length of the character strings in the file or its hash value (e.g., filesize), which can be acquired from the file system attribute data of the file, the FTCS object acquires the information without actually reading the file. The FTCS object is returned by either PHP extensions [1] or standard built-in extensions such as *file_get_contents*. The operations for an FTCS are of two types, lazy operations and eager operations. A lazy operation such as *echo* does not actually perform any file I/O operation, but instead the Web server sends the content directly to the client using the *sendfile* system call. An eager operation such as *preg_match* needs eager processing on the FTCS object.

3.2.2 Enhancement of the FastCGI Protocol and the Web Server

In cases where a PHP runtime creates an FTCS object but performs no actual I/O processing, the runtime can communicate with its Web server using the FastCGI protocol with our newly added header named X-ZeroCopy. File name and location are included in the body portion of a FastCGI message. The offset value indicating the position of the first character of the file name, and the length of the file name are written in an *X-ZeroCopy* header. The X-ZeroCopy header follows the HTTL extension header [34] and is specified as follows:

X-ZeroCopy = "X-ZeroCopy" ":" # (offset "/" length)

The header is recursively defined to allow the transmission of multiple files. For example, consider the following pseudo-FastCGI packet. The two files, /tmp/A.htm and /tmp/B.htm, are to be sent by a Web server via a zero-copy data transfer such with the *sendfile* system call. The second line indicates the body content of a FastCGI packet, and the first line is an X-ZeroCopy header that specifies the location of file references in the body by specifying the offsets and lengths of any file names.

```
X-ZeroCopy:5/10:21/10
hello/tmp/A.htmworld/tmp/B.htmbye
```

Our PHP runtime automatically generates this header and body content in a transparent manner from unmodified PHP scripts. The key feature of this approach is to allow a mix of dynamic content and the metadata for multiple files to be sent alternatively by the *sendfile* system call. For example, the original PHP script that is converted to the above FastCGI message by our PHP runtime is shown as follows.

[1] In PHP, an external C library function is called an "extension".

```
<?php
echo "hello";
echo file_get_contents('/tmp/A.html');
echo "world";
echo file_get_contents('/tmp/B.htm');
echo "bye";
?>
```

The PHP runtime first parses the PHP script and translates it into instruction code. Then the PHP runtime executes the translated instruction code. While executing the instruction code corresponding to *ECHO* (display a character string in the standard output), the PHP runtime determines whether or not the target variable is an FTCS object. In the current example, since the *file_get_content* extension returns an FTCS object, the FastCGI packet will contain the file's URI (e.g., */tmp/A.htm*) rather than the actual file contents.

The FastCGI packet generated by the PHP runtime is transmitted to the HTTP server for processing. The HTTP server parses the header of the received FastCGI packet. Upon finding an *X-ZeroCopy* header, the HTTP server returns the files designated by the *X-ZeroCopy* header to the client by using multiple *sendfile* system calls, instead of just forwarding the FastCGI packet to the socket. With this approach, a file that does not require processing by the PHP runtime is acquired directly by the HTTP server instead of passing through the PHP runtime, and is then transmitted to the client. Therefore the memory copying by the PHP runtime for the file can be omitted and the file is not copied, even during the memory copying carried out when the PHP runtime transmits the FastCGI packet to the HTTP server. Also, when the HTTP server returns its response to the client, the file is transmitted directly from the kernel memory to the client without passing through user memory. Therefore additional memory copying for the file is avoided, thus improving the operational performance of the Web application server. The interaction flow between a Web server and a PHP runtime in the case of sending a single file is depicted in Figure 2. The memory copying for C3 is eliminated, and C4 and C5 is reduced to C4' and C5' which only contain dynamically generated parts. We can calculate the amount of memory copied as C3 + (C4 – C4') + (C5 – C5') + (C6 – (C6'+C6'')).

3.2.3 File Change Management

When a Web application server uses this approach, there is a delay from the time when the PHP runtime executes the instruction code until the time when the HTTP server acquires the file specified by the instruction code. Therefore, the content of the file could change during this delay. Some assurance is needed that the file to be transmitted has the same content as it had at the time the code was executed by the PHP runtime. Our approach tracks the content of each file by using a hard link with a *Copy-on-write* function at the OS level or at the PHP processor level. For the OS-level approach, the PHP runtime generates a copy when the write operation is performed on a file, with a hard link using the *Copy-on-write* function implemented in the OS. This approach can only be used with a file system such as ext3cow [16]. For the PHP-level approach, the check is provided by an extension library for the PHP runtime. The PHP runtime locks the file with an exclusive lock, generates a hard link to a copy of the file, monitors any write to the file, and generates a copy when a write operation takes place, and unlocks the file after the copy has been transmitted to the HTTP server.

In general, it is best if the write operation is performed on the file before the transmission of the file by the HTTP server, so the PHP runtime generates and retains a copy of the file on which the write operation has not yet been performed. Then the HTTP server transmits the file for which the write operation has not yet been performed.

3.3 Implementation

We implemented our approach using our own PHP runtime, by modifying the Lighttpd HTTP server, and also by extending OpenMarket's FastCGI implementation [5] of the *mod_fastcgi* module to support a modified FastCGI protocol enhanced with our proposed HTTP header.

For our own language runtime, we extended our PHP runtime, known as P9 [28], a PHP runtime engine with a just-in-time compiler. A unique feature of our runtime is that it reuses an optimizing JIT compiler from a production Java virtual machine. Our motivation for that approach is to explore the retargeting of a production compiler for a statically typed language (Java) to a dynamically typed language (PHP). By exploiting various optimization features, we should be able to exploit many of the existing optimizations and features in the JIT compiler that were intended for a statically typed language. We also added new optimizations that are critical for optimizing a dynamically typed language.

4. Performance Evaluation

This section evaluates our optimization approach by running a set of micro-benchmarks and a standard Web benchmark application, SPECweb2005 [14].

4.1 Experimental Environment

All of the experiments in the following subsections use the same experimental environment described here. We used Lighttpd 1.4.19 as a web server running on an IBM IntelliStation M Pro 3.4-GHz Pentium 4 uniprocessor with 2 GB of RAM running Fedora Core 7 (kernel 2.6.17). Lighttpd was configured to use one parent process, two worker processes, and eight FastCGI processes. (We measured other variations of numbers of parent, worker, and FastCGI processes, but did not find other configurations to perform significantly better.) For the client machine, we used an IBM IntelliStation M Pro 2.0-GHz uniprocessor machine connected to the server via a 1 GB Ethernet LAN. The network latency was 0.12 ms, and the actual network throughput measured with *netperf* [30] was 941 Mbit/second.

4.2 Micro-benchmark

Theoretically our approach should be effective when the FastCGI communication overhead between the PHP runtime and the Web server is a major performance bottleneck. To find the threshold where file size becomes a bottleneck, we prepared a simple PHP script micro-benchmark that simply displays a file via the *file_get_contents* PHP extension. With the script, we evaluated the performance with file sizes ranging from 10 KB to 200 KB, as shown in Figure 3. The x-axis is the file size and the left y-axis shows the throughput of unmodified P9 and P9 with our optimization approach (P9ZC). Throughput is measured by the number of requests handled, and the right axis shows the speed-up ratio for P9 over P9ZC as measured by the *ab*

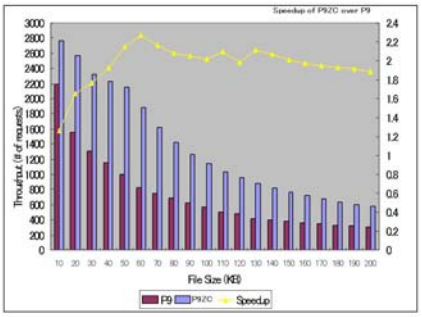
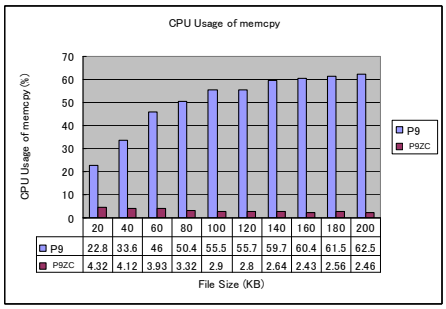


Figure 3. Throughput and Speedup (%) of P9ZC over P9 with varying file sizes.



	20	40	60	80	100	120	140	160	180	200
P9	22.8	33.6	46	50.4	55.5	55.7	59.7	60.4	61.5	62.5
P9ZC	4.32	4.12	3.93	3.32	2.9	2.8	2.64	2.43	2.56	2.46

Figure 4. CPU usage for the *memcpy* function.

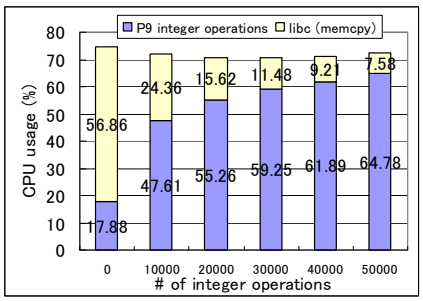


Figure 5. CPU usage for *memcpy* with varying numbers of integer operations and a 60-KB file.

(Apache Bench) tool with 1 process, 100 concurrent requests, and a 60 second run, measured after sufficient warm-up.

Figure 3 shows that the speedup of P9ZC over P9 increases from 1.26 for a 10 K file up to 2.26 for a 60 K file. After 60K, the speedup gradually decreases but P9ZC remains roughly twice as fast as P9. Figure 4 shows the cumulative CPU usage of the *memcpy* function used by the Lighttpd Web server and the PHP runtime, and the table below the graph shows the values as percentages of CPU usage. The graph shows that P9ZC significantly reduces CPU usage for the *memcpy* function while P9 uses approximately 20% to 60% of CPU time to perform memory copying. Figure 4 shows why performance improvements are not seen for large files with P9ZC in Figure 3. This is because the CPU usage of the *memcpy* function for P9 is saturated when file size is greater than 60KB. As with SPECweb2005 described in the next section, this is not a CPU-bound activity

Since these micro-benchmarks are idealized test cases for our optimization, we also performed experiments by adding multiple integer operations other than printing out a 60−KB file by using the *file_get_contents* extension to improve the CPU usage for further processing. By increasing the number of integer operations from 0 to 50,000, the CPU usage for the *memcpy* function was reduced by 56 percentage points to 7.6% as shown in Figure 5. The throughput and speedup (%) of P9ZC over P9 with varying numbers of integer operations is shown in Figure 6, and the result shows that the speedup gradually decreases, and in proportion to the percentage of the CPU used for the *memcpy* function.

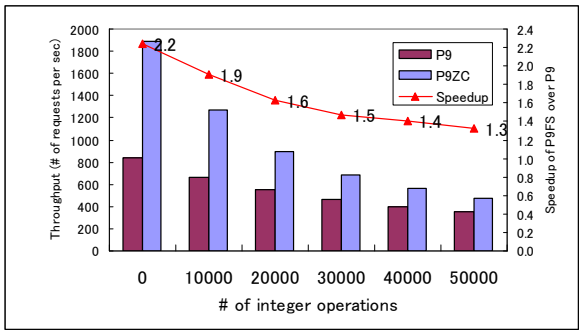


Figure 6. Throughput and Speedup (%) of P9ZC over P9 with varying numbers of integer operations and a 60-KB file.

4.3 SPECweb2005 Benchmark

SPECweb2005 [14] is a standard Web application benchmark defined by SPEC (Standard Performance Evaluation Corporation). The three-tier architecture used for SPECweb2005 consists of a Web server, a PHP runtime, and a BESIM backend database simulator. SPECweb2005 has three representative Web application scenarios: Banking, Ecommerce, and Support. Each scenario has different performance characteristics representing the different types of Web applications. The goal of SPECweb is not only to compare the numbers of request being processed, the throughput, but also to consider the response times. The completed requests are classified as good, tolerable, or failed depending on the response time set for each scenario. We will explain the performance results with each scenario. We use Zend, P9, and P9ZC, respectively, as labels for the type of PHP runtime. Zend is a PHP runtime available from [9] and configured with APC (Alternative PHP Cache) turned on, which allows the runtime to cache PHP intermediate codes. P9ZC is the enhanced PHP runtime that adds our proposed technique to the P9 runtime.

4.3.1 Banking Scenario

The banking scenario represents an online Web banking application, characterized as a secure Web application with SSL communication, which supports viewing account information, transferring money to other accounts, and so forth. The average size of the data sent to the Web client is 34.8 KB, as shown in Table 2. Figure 7 illustrates the performance results of the banking scenario in SPECweb with different runtime configurations, Zend, P9, and P9ZC. As shown in the graph, the peak performance of Zend is 1,000 sessions, versus 1,200 sessions for P9 and P9ZC. The performance advantage of P9 and P9ZC is attributable to just-in-time compilation, while Zend is an interpreter-based runtime. The results also show that there is no performance advantage of P9ZC over P9. Figure 24 shows that the memory copying operation consumes only around 2% of the CPU usage. This is because SSL processing accounts for a large portion of the CPU processing in this scenario, and thus there are few copy cycles for our approach to eliminate. Another important fact is that the *sendfile* system call becomes ineffective when using SSL. With an SSL-enabled configuration, the Web server loads the content of a file, encodes it, and then sends it to the network socket. This issue could be resolved by applying Keromytis's work [24] on an SSL-enabled *sendfile* that performs the SSL processing at the kernel level. Although SSL is

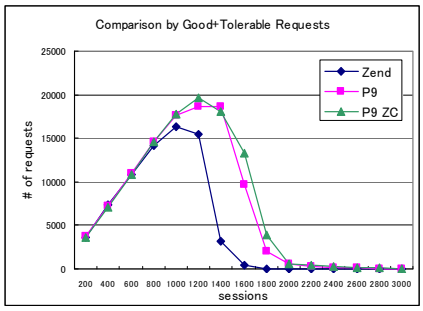


Figure 7. Comparison by Good+Tolerable Requests in Banking (SSL)

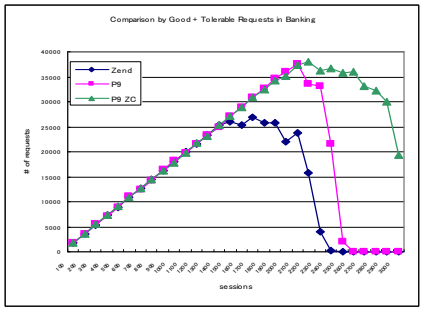


Figure 8. Good+Tolerable Requests in Banking (No SSL)

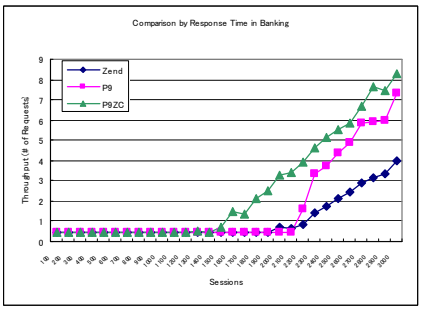


Figure 9. Comparison by Response Time in Banking (No SSL)

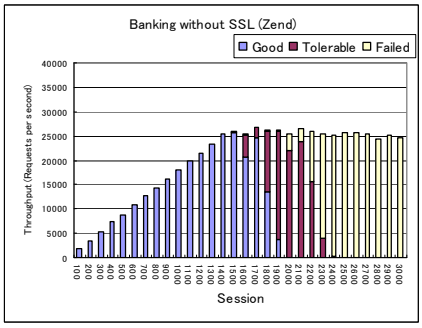


Figure 10. Performance of Zend in Banking (No SSL)

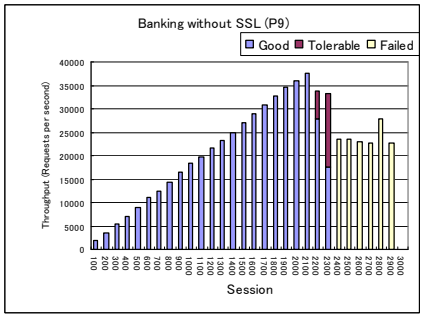


Figure 11. Performance of P9 in Banking (No SSL)

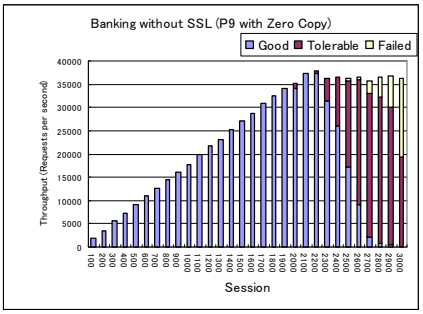


Figure 12. Performance of P9ZC in Banking (No SSL)

required for a compliant SPECweb benchmark, we also performed the experiment with SSL turned off to assess the effectiveness of our approach. In that experiment, we observed that the peak performance of P9ZC is 2,100 sessions, which is 5% better than the P9 peak of 2,000 sessions.

The performance advantage is more evident when considering the number of requests with good response times, as shown in Figure 11. A comparison by response time is shown in Figure 12. The response time of P9 increases more sharply from 2,200 sessions than P9ZC, and this phenomenon is also seen in the throughput data shown in Figure 11. CPU usage for memory copying operations is shown in Figure 24 as "Banking (No SSL)". P9 uses 7.2% of CPU `time` for memory copying, but P9ZC uses 5.1%. This 2.2% memory copying reduction contributes to the end-to-end performance improvement of P9ZC.

4.3.2 Ecommerce Scenario

The ecommerce scenario represents an online shopping application which supports searching for certain products, displaying product details, and finally purchasing the product. SSL is used only at checkout time. The average size of the data sent in this scenario is around 143.9 KB, which is larger than the other scenarios. Figure 13, Figure 14, and Figure 15 illustrate the performance results of the Ecommerce scenario with Zend, P9, and P9ZC, respectively. The peak performance of Zend is between 1,300 and 1,400 sessions and P9 is between 1,700 and 1,800 sessions. P9ZC reaches peak performance between 2,100 and 2,200 sessions. This result demonstrates that our optimization approach outperforms the original runtime, P9 by 22.2%, and Zend by 57.1%. We also compare the requests with good response times in Figure 16**.** This graph clearly shows the performance advantage of P9ZC. The comparison by response times is shown in Figure 17. Figure 17 also shows that the response time of P9ZC is better than the other two configurations at high load levels. Figure 24 displays the CPU usage for memory copying, showing that it drops from 15.6% to 3.4%. This memory reduction is larger than the other two scenarios since the volume of data traffic is larger, as previously mentioned.

4.3.3 Support Scenario

The support scenario represents a company support website where customers can download files such as drivers and manuals. The dynamic proportion is relatively small, and many files are simply sent from a Web server without any intervention by the PHP runtime. The average size of a data transfer is 78.5 KB. Figure 19, Figure 20, and Figure 21 illustrate the performance of the support scenario with Zend, P9, and P9ZC, respectively. The peak performance of Zend is between 700 and 800 sessions. P9 is between 800 and 900 sessions. P9ZC peaks between 1,200 and 1,300 sessions. This result demonstrates that our optimization outperforms P9 by 44.4%. The CPU usage graph in Figure 24 shows that the CPU usage for memory copying drops from 12.4% to 4.4%. This shows our approach is highly effective with the ecommerce and support scenarios, and could be effective with the banking scenario if SSL is not used. Since it is not realistic to send unencrypted data in a banking scenario, we will discuss this issue in Section 5. A comparison of the numbers of total requests is shown in Figure 18, and more detailed memory reduction information is shown in Table 2. This profiling data was measured when each runtime configuration (P9 and P9ZC) was operating at peak load.

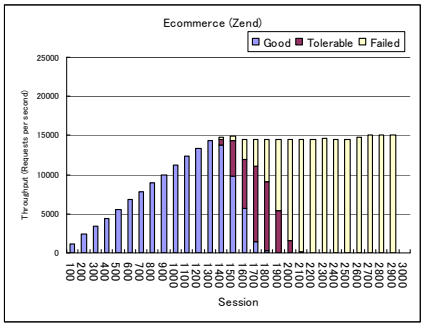


Figure 13. Performance of Zend in Ecommerce

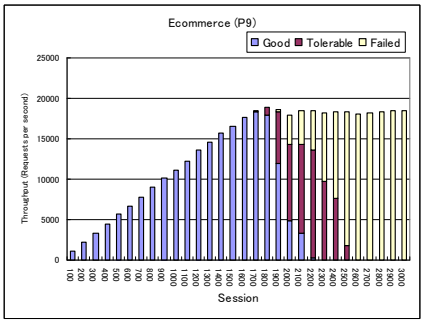


Figure 14. Performance of P9 in Ecommerce

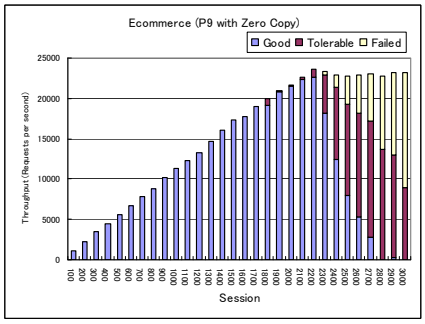


Figure 15. Performance of P9ZC in Ecommerce

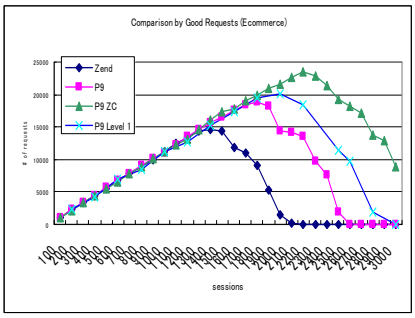


Figure 16. Comparison by Good Requests in Ecommerce

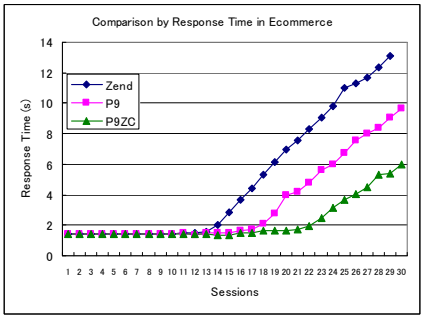


Figure 17. Comparison by Response Time

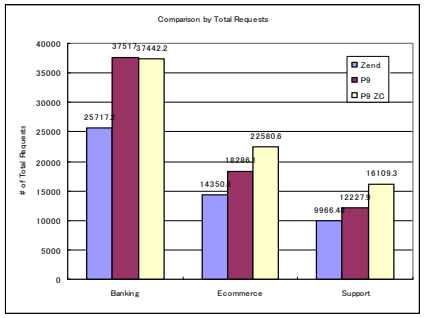


Figure 18. Comparison by Total Requests in 3 scenarios

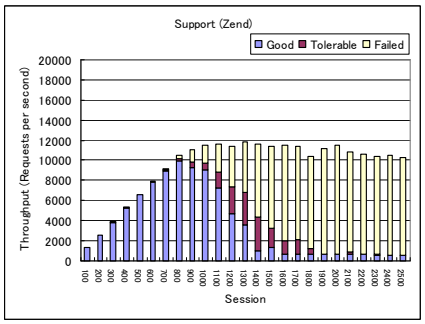


Figure 19. Performance of Zend in Support

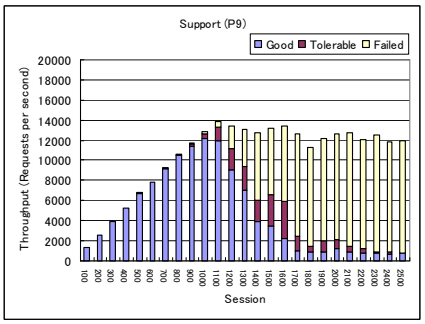


Figure 20. Performance of P9 in Support

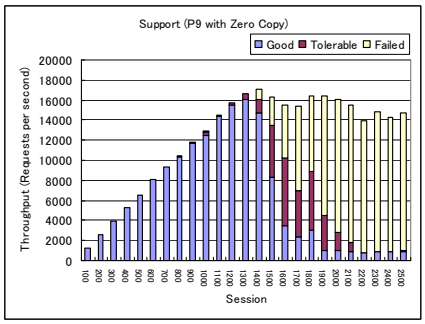


Figure 21. Performance of P9ZC in Support

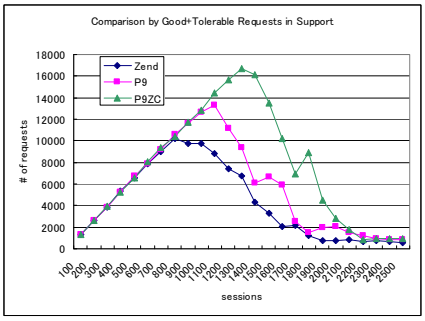


Figure 22. Comparison by Good+Tolerable Reqs in Support

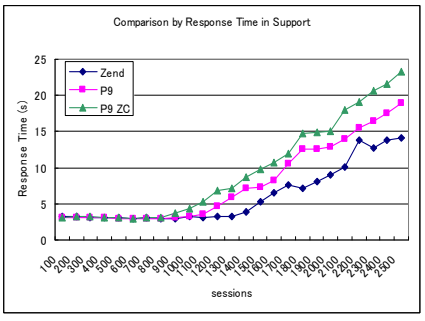


Figure 23. Comparison by Response Time in Support

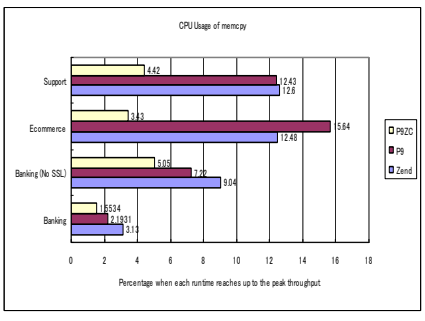


Figure 24. CPU usage of *memcpy* in 3 scenarios

4.3.4 Temporary Files Experiment

We also evaluated the effectiveness of storing the PHP runtime results in a temporary RAM disk file. Filename information was sent to a Web server with the special HTTP header, X-Sendfile as supported by the Lighttpd Web server, so we were able to reduce the amount of data transmitted between the PHP runtime and Web server. The Web server can directly send a given file via *sendfile* without copying it to user space. However, the file needs to be retrieved from kernel space or disk cache to a user-level buffer in the PHP runtime, which leads to extra memory overhead. Figure

16 shows the performance results of this approach, labeled "P9 Level1." Due to the reduction of data transmitted between the PHP runtime and Web server, and also the reduction of memory copying overhead in the Web server, the peak performance of P9 Level 1 is around 2,000 sessions, an 11% improvement compared to P9. We have only tested this approach with the Ecommerce scenario, but it shows that our approach, labeled P9ZC, is more effective than P9 Level 1.

5. Discussion

In this paper we focused on applying our optimization approach to PHP, but our approach could be used with other programming languages. For instance, dynamic scripting languages such as Ruby and Python are candidates, since their high-level nature is similar to PHP in using a lazy string implementation that is transparent to application programs. Ruby [22] allows an override of the *println* function and provides a polymorphic type of string, as well as file strings at the user-level, so we could implement our approach easily in a Ruby library. Also, the *IO.read* function in Ruby can read an entire file, similar to the *file_get_contents* function in PHP. For statically typed languages such as Java (used with JSP), we could use Tomcat and the Java Servlet Engine with Lighttpd using mod_proxy, which transfers the HTTP messages. The *transferTo* method [25] in java.nio.channels.FileChannel is available for transferring data from a channel to a given writable byte channel through the *sendfile* system call. This would allow us to reduce the communication overhead between the Java Servlet Engine and the Web server with our proposed approach.

We could also apply our technique to other protocol-based communication methods such as SCGI (Simple Common Gateway Interface) [13]. The proposed approach assumes the availability of UNIX domain sockets for inter-process communication, but could easily be implemented for generic TCP/IP communication.

Our technique could be extended to more general cases. If a compiler identifies a relatively long and constant string that may become a performance bottleneck, the compiler can automatically generate a file that contains the string at compilation time, and the compiler replaces that string with the filename of the generated file. The PHP runtime would specify offset and length information for the file with a X-ZeroCopy header, and then a Web server can handle the file via a zero-copy system call. The threshold for whether or not the compiler generates a file can be determined by prior benchmarks conducted on the target system environment, such as our micro-benchmarks described in Section 4.2.

In addition, we can extend our approach to the *include* statement in PHP. The include statement is used for including both static files and PHP scripts, but PHP developers often use this statement for including header and footer HTML files or cached files. Our PHP runtime can parse the included file, and if the runtime determines it is a static file, then it can replace it with an FTCS object, and send it with the *sendfile* system call.

6. Related Work

There are many papers that focus on improving the performance of individual software components of Web servers and language runtimes. For the ultimate performance gains, in-kernel Web servers [21][10] have been proposed to avoid expensive passing of data and control between kernel and user space. Robert et al. [10] uses SPECweb96 to measure the serving of static files with user-level Web servers and kernel-level Web servers, and shows that the fastest user-mode server measured (Zeus) is 3.6 times slower than the fastest kernel-mode server (AFPA).

Nahum et al. [1] evaluates the performance advantages of the *sendfile* system call. However as already mentioned, this technique only focuses on optimization when serving static HTML files. Our proposed approach uses *sendfile*, but it goes beyond the system call to consider what is required to support a programming language by extending the FastCGI protocol.

An approach known as Faster FastCGI [8] extends FastCGI protocol to send static files via the *sendfile* system call. This approach is available in the Lighttpd 1.5 beta HTTP server. In this approach, a PHP processor writes an execution result to a file assigned to shared memory (/dev/shm), and passes the file name as an HTTP header (X-Sendfile) in the FastCGI protocol. Thus, the FastCGI module (mod_php) of the HTTP server transmits each message with a *sendfile* system call. As described in Section 4.3.4, we also implemented and measured this method as "P9 Level 1", with the experimental results shown in Figure 16, and demonstrated that our method is more effective.

Generally zero copying is well known technique, and various layers from hardware to software apply this technique to reduce memory copying overhead. For instance, Ishikawa et al [26] proposes a zero copy approach to MPI (Message Passing Interface) communication layer. However, no attempts have done to apply this technique to Web application servers with multiple cooperative software components.

Keromytis et al. [24] propose an approach for enabling the *sendfile* system call in an SSL configuration by transferring data directly from disk to a cryptographic accelerator card, and then directly from the crypto card to the network card using OCF (OpenBSD Cryptographic Framework) kernel API. We could leverage their work to make our proposed method effective when SSL communication is required such as in the SPECweb Banking scenario.

Another paper from our team [32] describes a different approach for improving web server throughput by offloading template processing to the client side and only sending variable portions to the client. Meanwhile, the proposed approach in this paper takes a more traditional approach in that messages returned from the Web server to the web client have completely the same HTML content.

7. Conclusions and Future Directions

In this paper, we proposed a novel approach that improves Web application performance by optimizing communications between a Web server and a programming language runtime.

Our experiments show that our technique outperforms our original PHP runtime by 126% with micro-benchmarks and by 44% with SPECweb2005. The profiling data demonstrates that performance gains are proportional to the reduced use of the *memcpy* operation, which approximately equals the size of the generated HTML data. Without our proposed optimization technique, our PHP runtime with Just-in-Time compilation still outperforms the interpreter-based Zend engine (with APC enabled), but only by 28%. This data also shows that the effect of our proposed technique is similar to that of Just-in-Time compilation if memory copying is the major bottleneck.

Future work includes enhancements of the proposed approach. For example, to widen the uses of this optimization, we could

generate files at compile time when string constants are known and the lengths of the strings are long. Another area to explore is the enhancement of kernel support for dynamic Web applications such as an SSL-enabled *sendfile* system call [24] to deal with the case seen in the SPECweb2005 banking scenario.

8. REFERENCES

[1] Erich Nahum, Tsipora Barzilai, Dilip D.Kandlur, Performance Issues in WWW Servers, *IEEE/ACM Transactions on Networking (TON)*, Volume 10, Issue 1, February, 2002

[2] Vivek S. Pai, Peter Druschel, et al. "IO-Lite: A Unified I/O Buffering and Caching System", *Third Symposium in Operating Systems Design and Implementation (OSDI)*, 1999

[3] Dragan Stancevic, Zero copy I: user-mode perspective**,** *Linux Journal*, Volume 3, Issue 105 (January 3), 2003

[4] Hsiao-keng Jerry Chu , Zero-copy TCP in Solaris, *Proceedings of the USENIX 1996 Annual Technical Conference*, 1996

[5] Mark R. Brown, "FastCGI: A High-Performance Gateway Interface", *Fifth International World Wide Web Conference*, 1996

[6] Mark R. Brown, "Understanding FastCGI Application Performance", http://www.fastcgi.com/

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

[8] Lighttpd, http://www.lighttpd.net/

[9] PHP, http://www.php.net/

[10] Robert B. King , Mark Russinovich T, John M. Tracey, High-Performance Memory-Based Web Servers: Kernel and User-Space Performance, *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001

[11] David Pariaq, Tim Brecht, et al., Comparing the Performance of Web Server Architectures, *ACM SIGOPS Operating Systems Review*, 2007

[12] Gaurav Banga, Peter Druschel, Jeffrey C. Mogul, Better operating system features for faster network servers, *Proc. Of the Workshop on Internet Server Performance 1999*

[13] SCGI: A Simple Common Gateway Interface Alternative, http://python.ca/scgi/protocol.txt

[14] SPECweb2005, http://www.spec.org/web2005/

[15] Khalil Amiri, Sanghyun Park, et al., "DBProxy: A dynamic data cache for Web applications" *ICDE 2003*, pages 821-831, 2003

[16] Z. N. J. Peterson and R. Burns: Ext3cow; A Time-Shifting File System for Regulatory Compliance, *ACM Transactions on Storage (TOS)*, Volume 1, Issue 2 (May 2005), Pages 190-212

[17] Moti N. Thadani, An Efficient Zero-Copy I/O Framework for Unix, Sun Microsystems Technical Report, May 1995

[18] Dong-Jae Kang, Young-Ho Kim, et al., Design and Implementation of Zero-Copy Data Path for Efficient File Transmission, *High Performance Computing and Communication*, Sep, 2006

[19] Netcraft, http://survey.netcraft.com/Reports/200806/

[20] Apache Web Server, http://httpd.apache.org/

[21] Armol Shukla, et al, Evaluating the performance of user-space and kernel-space Web servers**,** *IBM Centre for Advanced Studies Conference*, 2004

[22] Ruby, http://www.ruby-lang.org/en/

[23] Scott Trent, Michiaki Tatsubori, Toyotaro Suzumura, Akihiko Tozawa, and Tamiya Onodera. Performance comparison of PHP and JSP as server-side scripting languages. *Proceedings of Middleware, 2008, ACM/IFIP/USENIX 9th International Middleware Conference*, Leuven, Belgium, December 1-5, 2008, pages 164–182. Springer, 2008.

[24] Angelos D. Keromytis et al., Cryptography as an operating system service: A case study, *ACM Transactions on Computer Systems (TOCS)*, Volume 24, Issue 1 (February 2006)

[25] Sathish K. Palaniappen, et al., Efficient Data Transfer through zero copy, IBM Developerworks, Sep, 2008, http://www.ibm.com/developerworks/library/j-zerocopy

[26] Francis O'Carroll, et Al., "The design and implementation of zero copy MPI using commodity hardware with a high performance network", *Proceedings of the 12th International Conference on Supercomputing*, 1998

[27] Nick Mitchell, Gary Sevitsky, Harini, Srivivasan, et al., The Diary of a Datum: An Approach to Modeling Runtime Complexity in Framework-Based Applications, IBM Research Report, RC23703, 2005

[28] Akihiko Tozawa, Michiaki Tatsubori, Scott Trent, Toyotaro Suzumura, and Tamiya Onodera, P9: High Performance PHP Runtime, *Japan Society for Software Science and Technology*, 25th Workshop, 2008

[29] OProfile, http://oprofile.sourceforge.net/

[30] Netperf, http://www.netperf.org/

[31] Lakshmish Ramaswamy, Ling Liu, and Fred Douglis Automatic Fragment Detection in Dynamic Web Pages and its Impact on Caching. *IEEE Transactions on Knowledge and Data Engineering* vol. 17 #6, 2005

[32] Michiaki Tatsubori and Toyotaro Suzumura, HTML Templates that Fly - A Template Engine Approach to Automated Offloading from Server to Client, WWW 2009 (*18th International World Wide Web Conference*), Spain Madrid, April, 2009

[33] Cecchet, E. Chanda, A., Elnikety, S., Marguerite, J., Zwaenepoel, W.: "Performance Comparison of Middleware Architectures for Generating Dynamic Web Content", *4th ACM/IFIP/USENIX International Middleware Conference* (2003)

[34] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee: Hypertext Transfer Protocol -- HTTP/1.1 (RFC 2616), June 1999.