MEMORY PERFORMANCE EVALUATION OF HIGH THROUGHPUT SERVERS

by

GARBA YA'U ISA

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

In Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

Dhahran, Saudi Arabia

June 2003

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by

GARBA YA'U ISA

under the direction of his Thesis Advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee

Dr. Abdul Waheed M. A. Sattar (Chairman)

Prof. Sadiq M. Sait (Member)

Dr. Farrukh M. Khan (Member)

Department Chairman
Prof. Sadiq M. Sait

Dean of Graduate Studies
Prof. Osama A. Jannadi

Date

To

My Parents and Entire Family

Acknowledgements

First and foremost, my gratitude to my Creator and Sustainer, Allah (SWT) for all the favors He has bestowed on me throughout my life. Peace and Blessings be upon His messenger, Muhammad (PBUH), the Seal of Prophets and Messengers. Million of thanks to my parents, brothers and sisters for all the prayer and support.

My utmost gratitude to my advisor, Dr. Abdul Waheed, for his guidance, enlightenment and encouragement throughout this work. My deepest gratitude to my thesis committee member and department chairman, Dr. Sadiq Sait, for the great support in this work and throughout my stay in the department. Also, my appreciation to Dr. Muhammad Farrukh Khan, for serving in my thesis committee and for the advice and guidance he offered.

Thanks to Balarabe Yusha'u of Mathematical Sciences Department, KFUPM, for his immeasurable help while I was seeking admission to KFUPM and during my academic expedition at KFUPM. Also, my thanks to the entire Nigerian community in KFUPM for all support. Immense gratitude to my PEL (Performance Engineering Laboratory) associates: Sarif, Amisu, Faheem, Sana'ullah and Sami. Thanks to Khalid El-Badawi of Information and Computer Science, for the informative discussions on C language coding and for translating the abstract to Arabic. Many thanks to my friends back in Nigeria, especially Hamisu Shehu Umar.

Acknowledgement is due to King Fahd University of Petroleum and Minerals, for providing the resources for this research.

Contents

List of Tables

List of Figures

THESIS ABSTRACT

Name:	Garba Ya'u Isa
Title:	Memory Performance Evaluation of High Throughput Servers
Major Field:	Computer Engineering
Date of Degree:	June 2003

High throughput of network infrastructure servers largely depends on their memory performance. While processor speed has been doubling roughly every eighteen months, memory access latencies reduce at a rate of about 10% per year. Bottleneck in server performance has been shifting from processor to cache, main memory and virtual memory performance. In this thesis, we analytically model memory access of transactions in three key high throughput servers: streaming media servers, web servers and software routers. We obtain optimistic peak throughputs of these servers for state-of-the-art general purpose processors with varying internal bus speeds. We also conduct a measurement-based performance evaluation of these high throughput servers. We identify memory subsystem as a potential performance bottleneck for streaming media servers and web servers, while context switching overhead and bus contention have greater impact on performance of software routers. To demonstrate how memory latency hiding can improve server performance, we design and implement a prototype RTP server that incorporates multithreading, and pre-fetching with buffering to hide memory (main and virtual) access latency. We evaluate the performance of our prototype against an RTP server, which does not incorporate above memory performance improvements and report that our prototype shows higher throughput with lower jitter.

MASTER OF SCIENCE DEGREE

King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia.

June 2003

خلاصة الرسالة

الإســـــــم : جاربا ياؤو عيسى

عنوان الرسالة: تقييم كفاءة الذاكرة في الخادمات ذات الإخراج العالي

التخصص: هندسة الحاسب الآلي

تاريخ التخرج: يونيو 2003م

تعتمد الخادمات ذات الطاقة العالية والتي تبنى منها الشبكة بدرجة عالية على كفاءة ذاكراتها. فبينما ازدادت سرعة المعالج إلى حوالي الضعف كل ثمانية عشر شهراً ، تتضاءل تأخر الوصول إلى الذاكرة إلى 10% في كل سنة تقريبا. فالإعاقة في كفاءة الخادم قد تحول من المعالج إلى كفاءة الذاكرة الفورية (cache) ، الذاكرة الرئيسية و الذاكرة الإفتراضية. في هذه الرسالة ، قمنا بصياغة تحليلية لوصول الذاكرة في ثلاث خادمات ذات اخراج عالي: خادمات الملتيميديا، خادمات الويب و موجهات (routers) البرامج. لقد حصلنا على الطاقة القصوى المتفائلة لهذه الخادمات لأحدث المعالجات ذات الأغراض المتعددة مع سرعات متغير للناقل الداخلي (internal bus). كما تناولنا تقييم الكفاءة على أساس القياس لهذه الخادمات ذات الطاقة العالية. ولقد تعلمنا أن نظام الذاكرة يمثل عائق كامن لكفاءة خادمات الملتيميديا وخادمات الويب ، بينما تمثل تحول السياق (context switching) ومقاومة الناقل العائق الأكبر لموجهات البرامج. حتى نبين مدى تأثير إخفاء الوصول إلى الذاكرة على تحسين أداء الخادم ، تم تصميم وتطبيق نموذج خادم أر تي بي (RTP) والذي يتضمن على تعدد الخيوط (multithreading) ، وطريقة الاستجلاب التمهيدي مع استخدام الذاكرة المساعده لإخفاء تأخر الوصول إلى الذاكرة (الرئيسية والإفتراضية). قيمنا أداء نموذجنا مع خادم آر تي بي والذي لم يظهر عليه أي تحسين في أداء الذاكرة و ذكرنا أن نموذجنا أظهر أخراجا عاليا مع تأخر منخفض.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن – الظهران

يونيو 2003م

Chapter 1

Introduction

High throughput of network infrastructure servers largely depends on their memory performance. While processor speed has been doubling roughly every eighteen months, memory access latencies reduce at a rate of about 10% per year. Bottleneck in server performance has shifted from processor to memory hierarchy, which includes cache, main memory and virtual memory performance.

This thesis is an outcome of our efforts to study the impact of memory subsystem performance on high throughput servers. It reports the results of our static analysis of memory access patterns and throughput of selected network applications. It also presents an in-depth measurement-based study of memory performance of three representative high throughput servers. In addition, a customized RTP server was implemented to investigate the benefits of memory latency hiding.

In this introductory chapter, we present a general background of this thesis.

We discuss memory subsystem issues related to the performance of high throughput servers. This discussion motivates the primary question that this thesis investigates: how on-chip cache, main memory, and disk can become significant performance bottleneck for high throughput servers in the context of network infrastructure applications.

1.1 High Throughput Servers

Growing use of the Internet requires high performance servers for such applications as World Wide Web (WWW) and real-time multimedia applications. Web servers, streaming servers, and software routers (henceforth, collectively termed as high throughput servers) are essentially high performance transaction processing engines that normally serve a large number of clients. The continuous growth of the Internet makes high throughput demands on these servers even more stringent; hence the performance of these servers must meet up with the demands in today's Internet applications and large number of clients.

1.1.1 Server Performance Issues

Due to growing interest in the development of novel technologies targeted at commerce and other critical applications, the performance of high throughput servers has become a key aspect in the design of information infrastructure. The objective

is to fulfill the growing requirement of offering access to an ever increasing volume of requests for information consisting of text, image, audio, and video from a large number of clients distributed across the Internet. Similarly, since its introduction in early 1990s, the concept of streaming media has experienced a dramatic growth and transformation from a novel technology into one of the mainstream manners in which people experience the Internet today. Indeed, such growth would not be possible without adequate progress in the development of various core technologies utilized by streaming media software and hardware.

Streaming servers need to retrieve media components in a synchronous fashion. These servers deliver live or on-demand audio or video content to potentially thousands of clients distributed across the Internet. Because of the stringent timing and quality-of-service requirements, high-bandwidth demands, and the CPU and memory intensive characteristics of these applications, the performance of the server hardware is critical for efficient performance and delivery of high quality multimedia contents.

Proxy server usage is growing and caching proxies have gained widespread deployment on the Internet. Frequent requests for a small number of popular objects have made caching highly successful in reducing server load, network congestion, and client perceived latency [1]. While most of the caching research to date has focused on caching of textual and image objects, streaming proxies are becoming increasingly popular. Caching streaming media objects with proxy servers poses

many new performance challenges [2]. The key challenge in designing such proxy servers is that they need to deal with heterogeneity in data characteristics as well as heterogeneity in the service requirements of applications.

1.1.2 Memory Hierarchy

There has been tremendous progress in microprocessor technology that leads to high speed CPUs. Also, advances in memory and magnetic disk technology have significantly improved memory and magnetic disk densities. However, memory and disk access and cycle times have lagged far behind improvements in their densities. Density of semiconductor DRAM and magnetic disks increase by approximately 50 - 60% per year, quadrupling in three years, but cycle time has improved very slowly, decreasing by about one-third in 10 years [3].

To alleviate the problem of widening performance gap between processor and main memory, computer architecture incorporates a hierarchical memory system in which data caches are widely used for hiding memory latency. Memory hierarchy is based on the principle of locality of reference – temporal and spatial. Temporal locality states that recently accessed data are likely to be accessed in the near future while spatial locality means that data whose addresses are near one another tend to be referenced close together in time [3]. Caches go a long way in improving performance for applications with small working data sets and large amounts of spatial and temporal locality. Often a small cache can provide enough storage to hold

most of the useful data required by the program at any time during its execution. If an application code is not tuned to exploit these locality characteristics, it may fail to achieve desired performance improvements.

1.2 Problem Statement

With current technology trends where processor-memory speed imbalance remains wide, tuning a program's memory performance has become increasingly important. Many research efforts have focused on improving the cache performance of scientific programs that use arrays as their primary data structure. Unfortunately, these techniques do not directly apply to high throughput servers. It is obvious that memory and disk overheads can inhibit the performance of any busy high throughput server. In this work, we are interested in the memory performance evaluation of high throughput servers to determine the specific conditions under which on-chip cache or memory becomes a major performance bottleneck for the server. Identifying these conditions is essential and will serve as a bed-rock for server design. This initial step will result in alleviating memory bottleneck and lead to improvement in overall performance. An efficient high throughput server must possess the following characteristics:

- Delivery of high throughput;
- Serve client transaction with minimal latency; and

- Responds to a large number of clients.

Memory performance issue in high throughput servers is a difficult problem for many reasons such as large amounts of data flowing through the CPU and memory system, mostly decreasing overall cache hit ratios and leading to a lot of memory copying. Writing code to optimize memory usage is also a complex task because of the nature of the data handled by these servers, which is difficult to tune for effective cache performance, since such data is hardly reusable and often the working set is large.

1.3 Contributions of this Work

This thesis addresses the memory performance problem that inhibits the performance of high throughput servers. The following are the major contributions of this thesis:

- Cache overhead analysis of various types of high throughput servers by static analysis and measurement-based experiments.
- Memory latency and bandwidth analysis and their impact on server throughput.
- Measurement-based performance evaluation of three representative network applications, namely web servers, streaming media servers, and software routers

involving Internet Protocol (IP) forwarding on general purpose computing platform.

- Design, implementation, and evaluation of a prototype streaming server, named Double Buffer RTP (DB-RTP) server.

These contributions are important for the state-of-the-art servers. For instance, understanding the role of memory performance on high throughput will give insight to a better design consideration. Addressing misconception of cache overheads for high data rate applications on general-purpose computing platform is also an essential design issue that must be addressed. Latency hiding is a viable work-around on the memory latency problem that can boost performance of applications.

1.4 Thesis Overview

The rest of this thesis is organized as follows. Chapter 2 provides background of high throughput servers, their performance tuning needs, evaluation methodologies, and related work. In Chapter 3, we analyze memory access pattern of network applications and calculate performance bounds based on cache and memory models. Chapter 4 reports our measurement-based memory performance study of key high throughput servers: streaming media servers, web servers, and software router for IP forwarding on general purpose computing hardware. In Chapter 5, we outline the design of our prototype streaming server that addresses some memory performance

issues. We also report our performance evaluation of the prototype RTP server that incorporates latency hiding. We finally draw conclusions about this research effort and outline the future direction of this work in Chapter 6.

Chapter 2

Background and Related Work

2.1 Introduction

Performance tuning of servers, especially those that are required to deliver high throughput is essential to meet up the demand of increasing number of clients requiring fast and efficient service. Servers that cannot deliver clients' request with acceptable quality of service (QoS) are likely to incur business lost.

In this chapter, we discuss background issues related with high throughput servers, their performance tuning and some design considerations to boost performance, and review related work in the literature focusing on high throughput servers and memory performance improvement techniques.

2.2 Server Performance Tuning

High throughput servers are often based on general-purpose computing platforms. Performance tuning is one of the basic design and deployment activities for these servers. In this section, we present the server performance tuning issues with respect to memory performance, latency hiding, multithreading, multiprocessing and clustering, and special architectures.

2.2.1 Memory Performance Tuning

Code transformation has been used intensively to improve memory performance [3]. Compiler controlled memory performance optimizations rely on source code analysis to identify target code blocks that can be transformed to improve data reuse. The objective of this process is to reduce the cache miss rate by improving data locality. We briefly review some popular techniques in this regard.

Array Padding

Since cache line sizes are often equal to a power-of-two value, array dimensions that are also a power-of-two cause unnecessary conflicts to occupy identical cache lines. Although a set-associative architecture reduces the contention due to multiple sets, the severity of the problem remains significant for larger applications due to a large number of memory accesses to an equally large number of arrays. A commonly used

technique to solve this problem is to pad the arrays to increase their dimensions by one [4]. Figure 2.1 shows an example of array padding in the declaration of an array, while the rest of the application code remains unchanged.

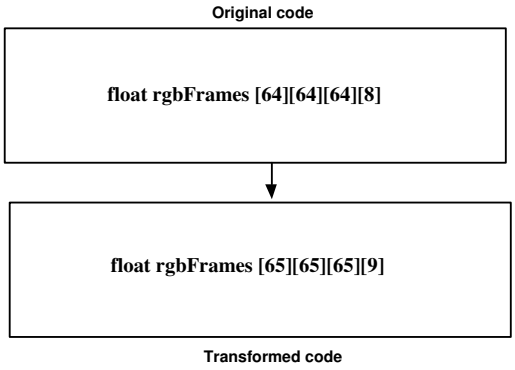


Figure 2.1: An example of using array padding to reduce conflict misses.

Array Restructuring

Minimization of strides of array references improves cache utilization. Unfortunately, there is no particular method of minimizing strides that can be applied in general to any given code. Array dimensions could be restructured so that the array elements that are used one after the other are stored in contiguous memory locations. Figure 2.2 shows the array declarations with their dimensions modified from the original code. The rest of the code remains unchanged.

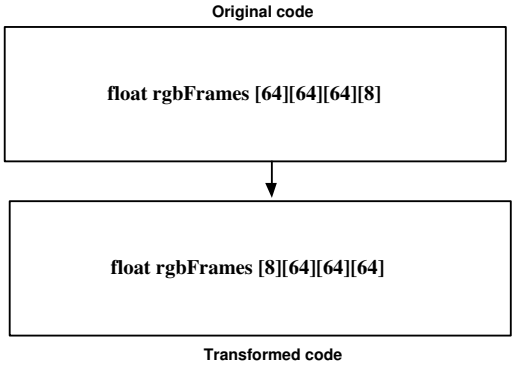


Figure 2.2: An array restructuring example to store the sequentially accessed data in contiguous memory locations.

Loop Nest Transformations

Loop nest transformation is another technique that can be used for stride minimization. With a nested loop, it is best to design the loop nest in such a way that subsequent memory access hit in contiguous locations or with small strides. Figure 2.3 presents an example of an original as well as modified code segment with loop nest transformations and array restructuring to minimize strides. In order to make such a transformation, the loop body needs to be analyzed to ensure that array accesses are independent of various loop indices to allow transforming the nest without changing the end result.

Reducing the overall memory consumption by an application decreases the overall cache misses. Blocking is another technique, which is often used to improve temporal locality. In blocking, instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks with a goal of maximizing

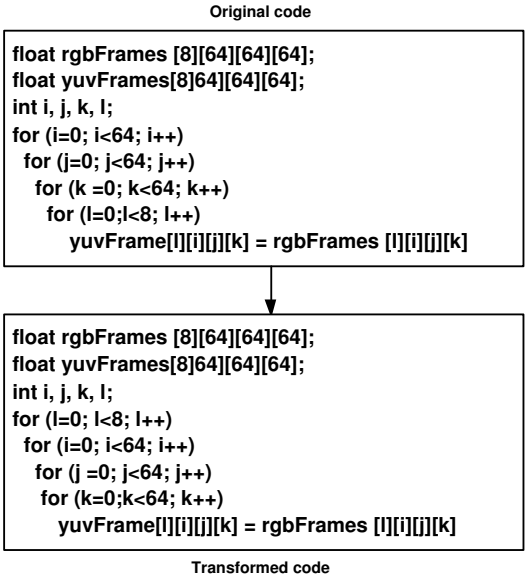


Figure 2.3: An example of loop nest modification with array restructuring to minimize strides.

accesses to the data loaded into cache before the data is replaced [3].

2.2.2 Latency Hiding and Multithreading

As memory access speed remains a technology issue that is unlikely to be resolved in the near future, several techniques are used to circumvent the latency constraints. One key technique is hiding memory access latency by implementing parallelism. Novel architectures and enhanced compilers could be used where a a processor can utilize the parallelism information to execute a large number of memory operations concurrently. For example, in multithreaded applications, when one thread is blocked due to memory latency, (e.g., a cache miss) or synchronization delay, the hardware switches to issuing instructions from another thread within a couple of

clock cycles [5].

Some explicitly parallel architectures that are capable of providing memory latency hiding are [5]:

- EPIC (IA-64) [6]: It exposes parallelism information to the hardware using very long word instructions (VLIW).
- VIRAM [7]: Expresses parallelism to hardware in the form of vector operations.
- Impulse [8]: Allows software to describe regular memory access patterns directly to the memory controller.

2.2.3 Multiprocessing and Clustering

Multiprocessing and clustering have been used in several applications to boost server performance. In fact, major web servers work in clusters or involve multiprocessor architectures. An example of a cluster implementation to boost performance is the Panama cluster router [9]. It includes a cluster of PCs connected by a high speed system area network. Panama has a decoupled system architecture that separates packet forwarding and packet computation paths. It derives performance improvement from parallelization of these tasks on the cluster.

2.2.4 Specialized Architectures

Another trend in server performance tuning is to employ special-purpose architectures that are specifically designed for a particular application. A typical example is a network processor. A network processor is a special-purpose programmable hardware specifically designed for network systems and applications [10]. Other special purpose architectures include ASICs (application specific integrated circuits) and data flow architectures. All specialized architectures have some common features that help them deliver high performance for their applications. These features include: flexibility through programmability, optimized architecture for specific target applications, and scalability with parallelism and pipelining.

2.3 Examples of High Throughput Servers

Due to a large number of users on the Internet, content distribution servers on the Internet are subjected to high traffic throughput and acceptable QoS demands. Some popular network core and edge applications on the internet today are: web, proxy, streaming servers, and routers. In this section, we review the design and performance issues relevant to these high throughput servers.

2.3.1 Web Servers

The purpose of a web server is to provide documents to WWW clients when they request for them. A web server operates in the following way. The server listens on a designated port (usually port 80) for a request from a WWW client to establish a TCP connection. Once a TCP connection is opened and the client has made its request, the server must respond to that request. The response includes a status code to inform the client if the request has succeeded or not. If the request is successful, a document is usually returned with the response. If the request is not successful, a reason for the failure is returned to the client.

2.3.2 Streaming Servers

An important issue in multimedia information systems, particularly for serving video and audio contents, is QoS guarantee. Streaming media servers better address this issue than web servers. To offer quality streaming services, servers are required to process and transmit data under timing constraints. A streaming media server typically consists of three subsystems, namely, a communicator (e.g., transport protocols), an operating system, and a storage system [11]. The operating system manages the essential resources, such as the CPU, main memory, storage, and all input and output devices. Since resources are limited, the server can only serve a limited number of clients with requested QoS. Therefore, resource management is

required to properly accommodate timing requirements.

2.3.3 Proxy Servers

A WWW proxy is an application program that accepts requests from a set of clients, forwards these requests to the appropriate servers (if required), and sends the requested data back to the clients. While receiving and serving requests from the clients, the proxy functions as a server. On the other hand, while forwarding requests to the origin servers, the proxy functions as a client. The proxies store frequently requested objects close to the clients in the hope of satisfying future client requests without contacting the origin servers. By keeping local copies of objects requested by clients, and using them to satisfy future requests for the same objects, caching proxies can reduce the amount of traffic flowing between clients and the origin servers. Proxy servers at the edges of networks are usually subjected to high transaction loads. Figure 2.4 depicts a typical proxy server setup.

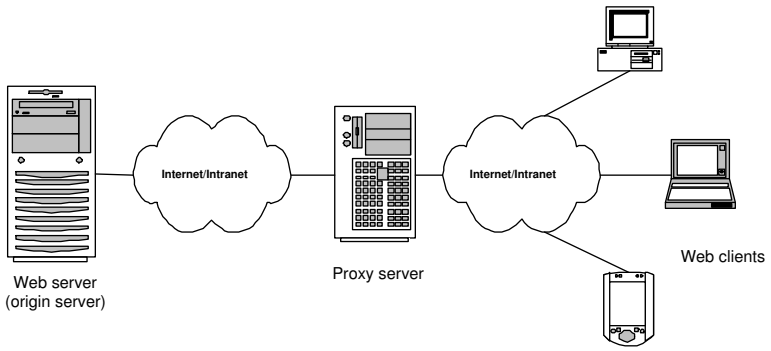


Figure 2.4: Proxy server setup consisting of an origin server, wide-area network, and edge of a network.

2.3.4 Software Routers and IP forwarding

Routers are network devices used at edges as well as core, responsible for routing and forwarding of packets. While routing is the process of building a routing table using one or more routing algorithms, forwarding is the process of moving a packet from an input port to an output port based on the destination of the packet. Forwarding is achieved in consultation with the routing table.

Decreasing cost of general-purpose computing hardware and their increasing performance has begun to attract researchers to consider the deployment of these general-purpose hardware for IP routing and forwarding purposes. IP packet forwarding is described in RFC 1812 [12]. The following steps are involved in forwarding an IP packet:

- Verify the header checksum. If cannot verify correctness, drop the packet without any further action;
- Check for IP options;
- Look up the destination address in the forwarding table and decide which output port packet should go;
- If no route is found, return ICMP Destination Unreachable;
- If the router itself (typically its control processor) is the packet's final destination, deliver it;

- Verify that the packet's time-to-live (TTL) is > 0. If check fails, return a (possibly rate-limited) ICMP error message ("Time exceeded");
- Decrement TTL;
- Update the header checksum (if done in an error-preserving way, the initial verification can be skipped);
- Verify whether the MTU of the outgoing interface is large enough; if not, fragment; and
- Send the packet to the appropriate output interface as determined by the forwarding lookup.

Four most important performance issues related to router performance are: (i) use of interrupts, (ii) bus bandwidth, (iii) speed of the CPU, and (iv) bandwidth of the memory. Time taken to process interrupts can be quite significant for high performance routing. Any amount of time taken between hardware generating an interrupt and reading relevant data is a direct contributor to latency within the router. Bus bandwidth in the router host is very important. The machine bus is the common communication channel that nearly all hardware components use to communicate. When attempting to route between a number of separate devices, it is possible for the bus to become a performance bottleneck due to contention. The CPU can also be a performance bottleneck especially when there is a lot of

processing due to firewall or NAT (network address translation) rules [13].

2.4 Evaluation Methodologies

System performance evaluation falls under one of the following methodologies: static analysis, simulation or measurement. In static analysis, mathematical equations representing a model of the system can provide insight on performance, especially bounds (lower or upper limits on performance). In static analysis, mostly accuracy is compromised to some extent for simplicity. Trace driven simulation technique is well-known for designing and analyzing processor caches. Instrumented applications are executed to obtain a trace of all memory accesses (load and store addresses with or without size of data transfer) in a benchmark program. This trace is consumed by a simulator that can predict the cache performance of a new architecture whose design parameters, such as cache line size, degree of associativity, capacity, etc., can be evaluated under such workload. Unlike the other two methods, measurement requires an existing system or a prototype. Measurement based evaluation provides the most realistic assessment of system performance and behavior, though it takes more time and is likely to be more expensive. Processor on-chip counters are frequently used to capture low level measurements of metrics like cache misses, instruction cycles, etc.

2.5 Related Work

In this section, we provide an overview of related work in the literature. We discuss research efforts towards improving memory performance using several techniques such as improving cache miss rate and use of memory compression. We also review work on performance evaluation of high throughput servers and video server design issues. We finally discuss work on routing and IP forwarding on general-purpose computing platform.

2.5.1 Improving Memory Performance

Improving cache miss rate has significant impact on memory system performance since high miss penalty will be minimized. Another approach for improving memory speed is by increasing the bandwidth, which will effectively improve memory access latency. Compressing memory pages minimize the tendency of swapping such pages to disk, hence minimizing slow disk access by keeping the compressed page in memory.

Performance of Cache and TLB

The most important performance parameters of a memory hierarchy are cache miss latency, TLB (Translation Lookaside Buffer) miss latency, and effective data path parallelism [14]. The substantial research efforts on improving the performance of

cache, memory, and TLB shows the extent of their importance. So many techniques have been proposed in the literature for improving cache performance by either reducing miss rates or miss penalty . Techniques for minimizing cache miss rate include use of larger cache block size and large cache capacity, higher associativity, use of victim caches, implementing pseudo-associative caches, hardware prefetching of instructions and data, compiler-controlled prefetching and compiler optimizations [3]. Cache miss penalty could also be reduced by: (1) giving priority to read misses over write misses; (2) use of early restart and critical word first; (3) use of nonblocking caches to reduce stalls on cache misses; and (4) use of second and third level caches [3]. Not surprisingly, most of the research efforts on improving cache performance are on numerical applications [15, 16, 17, 18]. Few cases have been reported on the study and optimizing cache performance for other applications like database systems [19, 20, 21, 22, 23]. Cache misses could be represented analytically, providing a general framework to guide code optimizations for improving cache performance. Cache miss equations have been used to determine array padding and offset amounts that minimize cache misses [24]. High throughput servers will benefit from some of these techniques like nonblocking caches that will reduce stalls on cache misses and use of larger cache block size since working set for the servers are mostly large.

Improving Main Memory Performance

Since reducing latency of memory has been a slow process, it is generally easier to improve memory bandwidth with novel organizations than it is to reduce memory access latency. Several techniques have also been reported in the literature for improving memory bandwidth . These techniques include (1) wider main memory, (2) simple interleaved memory to take advantage of the potential parallelism of having many DRAMs in a memory system, (3) independent memory banks that will allow multiple independent accesses, (4) avoiding memory bank conflicts, and (5) DRAM-specific interleaving [3]. High throughput servers that require transfer of large block of data will benefit from a wider main memory, effectively minimizing memory access latency. The parallelism inherent in memory interleaving is also good for these servers.

System Software Modifications

Several memory performance tuning approaches are based on modifying operating system, compiler, or even application source code. Recent work on memory performance tuning has proposed compressing memory pages in preference to swapping them out to disk [25, 26, 27, 28, 29]. The goal of memory compression is to hide the disk latencies by storing swapped out page frames in a compressed form, while residing in physical memory. On a subsequent page fault, the page can be quickly decompressed and supplied to the application program. Roy et al [30] implemented

a compressed memory on a Linux operating system. Their implementation is in the form of a loadable device driver, which can simply be unloaded for those applications that do not benefit from memory compression. The implementation shows speed-ups ranging from 5% to 250% on SPEC 2000 benchmarks and computational kernel applications. Researchers working on the Impulse Project [31, 32] introduced an optional level of address translation at the memory controller. This feature exploits "unused" physical addresses that can be translated to "real" physical addresses at the memory controller. Using Impulse requires modifications to applications (or compilers) and operating systems, but requires no hardware modifications to processors, caches, or buses. Impulse can reportedly speed-up a range of applications from 20% to over a factor of 5. Memory compression may not be a suitable alternative for high throughput servers since reusability of data is minimal. Keeping a compressed data in memory that is likely not be used again is not an advantage.

2.5.2 Performance Evaluation of High Throughput Servers

We present performance evaluation methodologies and performance analysis of high throughput servers. We outline some video server design issues. We similarly review research efforts on software routing and IP forwarding on general-purpose computing platform.

Streaming Media Servers

Performance of a streaming server is a key factor contributing to the quality of the multimedia content for the end-users. Shenoy et al [33] highlighted some fundamental issues arising in multimedia server design. Technical challenges in design such as storage and retrieval of multiresolution data, scalability and management were presented. Sohn et al [34] evaluated the performance of a small-scale Video on Demand (VOD) server. They conducted a measurement-based study in which they outlined the predictability of the real-time scheduler and the performance of the VOD server. Results of the performance measurements showed that the network protocol processing is a source of non-predictability. They found that high performance processor should be used to process the network protocol. However, the performance of the storage system was not a problem to the VOD service.

A significant amount of work is reported in the literature on the disk storage performance for streaming media servers. Due to large volumes of video and other multimedia content, storage and retrieval techniques play an important role in the performance of the server. A storage hierarchy to design a low-cost cache for a movie on demand (MOD) server was proposed in [35]. The hierarchy consists of a disk, which stores the popular movies, and a small amount of RAM buffers that store only part of the movies. Due to low cost of disks, the cost of a MOD server based on the proposed architecture is substantially lower than one in which the entire movie

is loaded into RAM. Another multimedia architecture and data retrieval model for supporting simultaneously multiple clients requesting files of different playback rates is presented in [36]. The performance of the architecture was investigated using a circular SCAN disk scheduling policy in terms of the maximum number of concurrent video streams it can support.

Some studies of multimedia servers pay attention to I/O subsystems due to high throughput demand of the servers. In fact, streaming media servers are often I/O bound. A study focused on the design of an I/O subsystem for a continuous media server is reported by Weeks et al [37]. They proposed several improved architectures based on an existing device: Intel i960RP I/O processor, and evaluated their performance. They reported that utilization of the I/O processor solved the main memory bottleneck problem, but created a new bottleneck in i960RP memory. I/O performance in multimedia servers has also been investigated using simulation [38]. Various I/O issues in multimedia systems have been discussed in [39], focusing on disk scheduling, SCSI bus contention and effect of buffer space on the performance of real-time requests and aperiodic requests.

Rixner [40] proposed the Imagine architecture for streaming media processor, which delivers a peak performance of 20 billion floating-point operations per second. Imagine efficiently supports 48 arithmetic units with a three-tiered data bandwidth hierarchy. At the base of this hierarchy, streaming media system employs memory access scheduling to maximize the sustained bandwidth of external DRAM. At the

center of the hierarchy, global stream register file enables streams of data to be recirculated directly from one computation kernel to the next without returning these data to memory. Also, local distributed register files that directly feed the arithmetic units enable temporary data to be stored locally so that it does not consume costly global register bandwidth. The bandwidth hierarchy enables Imagine to achieve up to 96% of the performance of a stream processor with infinite memory bandwidth from memory and the global register file.

Design of Video Servers

Several designs and architectures of video servers have been reported as improving one or even some aspect of performance of the servers. Bulk of the work however addresses the issue of storage and disk performance. Issues in multimedia server design are presented in [33].

The fundamental design issue is that of addressing quality of service constraints like delay, jitter and packet loss while also enhancing the capability of the server to handle large number of concurrent streams of video. Table 2.1 outlines some techniques in the design of video servers.

Table 2.1: Some design issues in media storage and retrieval techniques.

	Techniques	**References**
Media storage	Data striping	[41, 11]
	Hierarchical storage	[11]
	Fault tolerance	[42, 43, 44, 45, 46]
Media retrieval	Buffering	[47, 48]
	Caching	[49, 47]
	Batching	[50, 51]

Web and Proxy Servers

Web servers are key part of the Internet infrastructure today since web traffic accounts for substantial proportion of the traffic on the Internet. There has been tremendous amount of work on web servers ranging from performance studies to workload characterization [52] and even security issues [53, 54]. But just like the case of streaming media servers, there is no significant work on the performance of cache and memory subsystem. Iyengar et al report [55] performance study focused on improving the performance of the web server in the situation where the CPU becomes the limiting resource. In [56] a new web server mechanism was reported. JAWS was designed as an object-oriented web server that was explicitly meant to alleviate the performance bottlenecks identified in existing web servers. The performance optimizations used in JAWS included adaptive pre-spawned threading, intelligent caching, and prioritized request processing. Performance results were presented showing the scalability and efficiency of the proposed design. This is an attempt to improve the performance of the web server itself and not the underlay-

ing hardware.

Comparative performance characteristics are often studied using web server benchmarking tools. A measurement based performance study of Apache and Microsoft Internet Information server is reported in [57]. Their study focused on comparative performance on same hardware, but no attention was paid on the impact of the underlaying hardware. Trecordi and Verger [58] studied the main factors affecting the performance and scalability of web servers. They take into account the impacts of the server software architecture, operating system and the underlaying server hardware. They reported numerical results that reveal that the performance and scalability of WWW servers heavily depend on a lot of parameters that should be properly tuned. Although this study discussed cache and virtual memory system, no measurement on any metric related to the cache and virtual memory was reported.

2.5.3 Software Routing and IP Forwarding

There is a renewed interest in software based routers. These routers are hosted on PC platforms. A software router provides the flexibility of supporting value-added services, such as firewalls, differentiated services, load-balancing, etc., in addition to routing and forwarding IP datagrams. The main challenge is to achieve line speeds like hardware routers by eliminating any performance bottlenecks. Simple nature of IP forwarding transactions is another reason why many researchers are working on software routers.

Click [59, 60] provides a software architecture for building flexible and configurable routers. The design of this framework is highly modular in which individual elements implement simple router functions like packet classification, queueing, scheduling and interfacing with network devices. When elements are connected into a graph, a complete configuration is formed with packets flowing along the graph's edges. Click's power comes from two specific features: pull processing and flow-based router context. The former models packet motion driven by transmitting intefaces and makes packet schedules easy to compose, while in the case of flow-based router context, the router graph is examined to help an element locate other interesting elements. The authors reported an implementation on a general purpose hardware. A Click IP router running on a 450 MHz Pentium III with Linux 2.2.10 can forward 73,000 64-byte packets per second and 250-byte packets at 100 megabits per second. An interesting aspect of Click is that it can both be implemented in operating system user and kernel space.

A performance model and simulation study of a PC based IP software router was reported in [61]. The model is an open multiclass priority network of queues, which was evaluated through simulation. The model estimated probability distribution function of packet latency. The validity and accuracy of the multiclass model was established by comparing both packet processing latency traces and their complementary cumulative probability functions. Though simulation might not capture several important aspect of software and hardware interaction in real implemen-

tation of such routers, the multiclass model is capable of estimating the packet-processing latency of a PC based IP software router at several levels of detail. This makes the multiclass model suitable for capacity planning of PC based IP software routers expected to support Quality of Service. Other studies of software routing and IP forwarding on general purpose hardware and custom hardware are reported in [62, 63].

Chapter 3

Analysis of Memory Accesses

3.1 Introduction

Analytical model provides us quick means to calculate expected performance numbers. With some simplification assumption, it is possible to compute performance values, which we can compare with measurement based values.

In this chapter, we use simple analytical model to abstract resources related to data movement in a typical transaction and we obtain throughput optimistic bounds for representative network applications: HTTP, RTP and IP forwarding. We compare these throughput with corresponding throughput obtained using measurement to capture CPU overhead.

3.2 Data Flow Issues

The architecture of a general purpose processor based computing platform is shown in Figure 3.1. Hardware resources on such a server include: processor, on-chip and/or off-chip low latency caches, main memory, one or more disks, and one or more network interfaces. The hardware resources are connected to one another through a high-bandwidth internal system bus, a low-bandwidth I/O bus, and a bus controller. In terms of data flow, both within as well as outside the server, there are four data transfer paths. These paths include: (1) CPU-memory data transfer of operands for operations that utilize CPU time for arithmetic and/or logical instruction executions; (2) memory-memory data transfers (that go through CPU) for copying blocks of data from one network protocol layer to another; (3) disk-memory data transfers through Direct Memory Access (DMA) for retrieving or storing large data; and (4) network interface-memory data transfers through DMA for incoming or outgoing data through the network. While the first and the second types of data transfer use internal bus, the third and fourth types of transfer utilize I/O bus.

3.3 Latency Model and Memory Overhead

Transactions performed by a typical network infrastructure server can be characterized by three activities: (1) reading an incoming transaction request from a memory

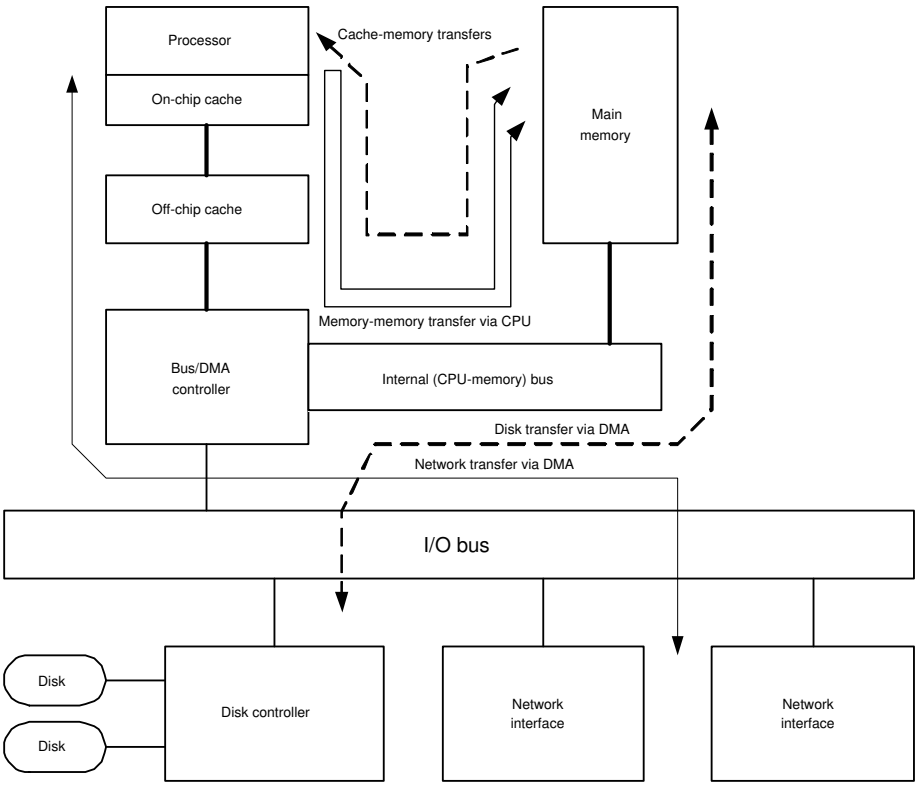


Figure 3.1: Architecture of a typical server built on a general-purpose platform with four data transfer paths.

location (network buffer) through a network interface; (2) request processing that requires CPU time; and (3) writing the response to a memory location (a network buffer) that results in outgoing data transfer through network interface. These activities are not necessarily performed in the same order. Also, one transaction may involve multiple operations of each one of the above three general categories of operations. The above discussion of hardware and software resources of a typical server allows us to consider all non-CPU operations as memory accesses of three types: (1) memory-CPU (or cache) transfers; (2) memory-memory transfers; and (3) memory-I/O and/or memory-network transfers. In the rest of this section, we determine the latencies due to each of these memory access operations.

3.3.1 Memory-CPU Transfers

Part of a network transaction utilizes CPU cycles for functions such as: decrementing time-to-live of an IP packet, computing checksum for an IP packet header, computing retransmission time-out value for a TCP segment, computing checksum of a TCP segment, etc. Not all of these computations require transferring every word of the PDU to the CPU from memory in a sequential order. Some protocol processing functions require updating a protocol header, which consists of a small number of bytes. However, some operations such as checksum calculation of an entire Protocol Data Unit (PDU) (e.g., a TCP segment) require sequential access to a contiguous block of memory locations. Due to multiple levels of memory hierarchy,

these contiguous data blocks are first transferred to cache from where CPU can access them. This process involves several memory stall cycles that contribute to the transaction latency. Memory stall cycles can be measured in terms of miss rate for an application [3], such that:

$$Memory\ stall\ cycles = (IC)(AR)(MR)(MP) \tag{3.1}$$

where IC represents *instruction count*, AR specifies memory *access rate* in terms of the number of memory accesses per instruction, MR is the *miss rate*, which is the ratio of cache misses to memory accesses, and MP specifies *miss penalty* in terms of clock cycles. Considering only data cache misses, we can further simplify the expression for memory stalls by assuming that each instruction includes one memory access, that is $AR = 1$. Then we can re-write the above expression as:

$$Memory\ stall\ cycles = (IC)(MR)(MP) \tag{3.2}$$

Using this expression, we can calculate the memory stall cycles through measurements to determine IC and MR while MP is known for every level of memory hierarchy. In order to get further insight into memory stalls, we can use the general observation that access to each subsequent level of memory hierarchy is slower by one order of magnitude. If access to L1 cache takes one clock cycle, we can assume that penalty for an L1 cache miss will be of the order of 10 clock cycles, which is

true for several processors. Since miss ratio MR is dependent on application characteristics, we can further analyze it by focusing on network applications. In the worst case, $MR = 1$ and using $MP = 10$, the number of memory stall cycles will be 10 times of IC (i.e., ten stalls per instruction), which is quite high. However, the situation is not as bad for network applications. Unlike computational applications, network PDUs do not contain repeatable data. Therefore, temporal locality does not exist in such data. However, contiguous data are accessed as a block (with a stride of 1) and spatial locality does exist. For instance, if an L1 data cache consists of 16 words (or 32 bytes), loading one word to a cache line will also bring 7 contiguous words into the cache that are to be used subsequently. Thus, effective value of $MR = \frac{1}{16}$ or 6.25% in this case. Memory stall cycles will be very close to the instruction count in such a case. Generally, $MR = \frac{1}{(\frac{L}{W})} = \frac{W}{L}$ where W is the width of each memory access (in bytes) and L is the length of each cache line (also in bytes). One important issue that needs to be analyzed is the role of caches for high-throughput network applications.

It is commonly believed that due to lack of temporal locality in network PDUs, data caches introduce unnecessary delays. It will be useful to calculate the exact amount of overhead introduced due to caches. One way to specify this overhead is to calculate the ratio of execution times (in terms of clock cycles) with and without a cache (or with no memory stall cycles). Execution time without cache can be expressed as:

$$(Execution\ time)_{no-cache} = (IC)(CPI)(CC) \tag{3.3}$$

where CPI represents average *clock cycles per instruction* and CC is the clock cycle time. Execution time with cache will result in memory stalls and can be given as:

$$(Execution\ time)_{with-cache} = (IC)(CPI)(CC)(1 + (MR)(MP)) \tag{3.4}$$

Thus the overhead of having a cache for a network application can be calculated as a ratio of two execution times as:

$$Cache\ overhead = 1 + (MR)(MP) = 1 + (10)(MR) \tag{3.5}$$

In the worst case, $MR = 1$ and cache will result in 11 times higher latency than an architecture that simply uses a fast memory without a cache. This rare case may occur when stride is such that every memory access results in a cache miss. Under such a worst-case scenario, latency of transferring a PDU from memory to CPU is determined by the bandwidth of the internal bus. The best case when $MR = 0$ is trivial and corresponds to transactions that do not involve any memory accesses. In such cases, cache does not introduce any additional latency. A more practical case occurs when MR is non-zero and typically close to 0.1. In such cases, the product

$(MR)(MP)$ approaches 1. That is, in practice latency introduced by a cache is as much as the ideal execution time without memory stalls. Therefore, using a general-purpose processor based server architecture may restrict the average throughput to half of what would have been possible in a special-purpose architecture without a data cache.

For the case where the PDU is copied from the memory to the CPU and if there is no contention for the bus, such transfer is simply limited by the bandwidth of internal bus. If the internal bus has a bandwidth of B_i MBytes/Sec and the cache line is 32 bytes, the latency to copy a block of S bytes is given as:

$$Memory - CPU\ latency = \frac{S}{32B_i}\ \mu sec \tag{3.6}$$

3.3.2 Memory-Memory Transfers

Protocol processing typically involves copying a block of contiguous (stride = 1) memory locations to a different location to pass a protocol data unit to the subsequent layer. If there is no contention for the bus, such transfers are simply limited by the bandwidth of internal bus. If the internal bus has a bandwidth of B_i MBytes/Sec, the latency to copy a block of S bytes is given as:

$$Memory - memory\ latency = \frac{2S}{B_i}\ \mu sec \tag{3.7}$$

Current generation of general-purpose processor based server architectures are capable of transferring multiple GBytes/Sec over the internal bus. For instance, a 2 GHz Pentium IV processor can allow up to 4 GBytes/Sec of data transfer over its internal bus. This is equivalent to 32 Gbits/Sec of data transfer rate within the server.

3.3.3 Memory-I/O and Memory-Network Transfers

Both I/O and network operations involve data movement over the I/O bus through a bridge. Therefore, both types of operations are similar from data transfer perspective. Every transaction starts and ends at network interface card (NIC), which is connected to the I/O bus. The I/O bus is typically slower compared to the internal bus. If bandwidth of the external bus is B_e MBytes/Sec, latency to pass a PDU of S bytes is given as:

$$Memory - network\ latency = \frac{S}{B_e}\ \mu sec \tag{3.8}$$

Both I/O and network operations use DMA controller to transfer data to or from memory without involving the processor.

3.4 Reference Applications

This thesis focuses on three high throughput network infrastructure applications: streaming media servers, web servers, and software routers. We apply memory latency calculations to these applications. We are interested in calculating the latencies of transactions of these three applications running on general-purpose processor based servers. Our goal is to identify the frequency of each of the four types of data transfer operations for every transaction. We assume that the latency of a transaction is the sum of the following latencies: processing by CPU, memory-CPU transfer, memory-memory copy, memory-NIC transfer, and memory-I/O transfer.

3.4.1 RTP Transaction Latency

Compressed video and audio transmission over the Internet uses streaming to allow the receiver to playback chunks of entire document as they arrive. Real Time Protocol (RTP) is used in conjunction with Real Time Control Protocol (RTCP) to deliver streaming media content. A streaming media server can store the content on the disk in multiple chunks that can be streamed on demand from a client after appending an RTP header to each one of them. Streaming is not restricted to transmission of stored audio or video only. It may also include live audio/video as well as other interactive applications, such as video conferencing. However, to keep our focus on high throughput streaming servers, we consider the case where chunks

of data are available in the main memory from where they can be streamed to the requesting client by appending RTP headers.

A complete streaming transaction has two parts: a request and a response that streams multiple RTP packets. As most of the data transfer is due to streaming of RTP packets, the request part is simply irrelevant to our calculations. Response part consists of several RTP packet transfer transactions. Each RTP streaming transaction consists of following operations:

1. Formation of an RTP packet consisting of a header and a chunk of compressed audio/video data frames. This RTP packet is copied to the transport layer (often using UDP rather than TCP). This involves one memory-memory data transfer.

2. Calculation of UDP (the same applies to TCP) segment checksum requires entire segment to visit CPU (word-by-word, sequentially) through cache. Thus there is one memory-CPU data transfer for entire transport layer PDU.

3. Transport PDU is copied to an IP buffer. This operation involves one memory-memory data transfer of entire PDU.

4. Finally, the IP packet is handed over to the NIC resulting in one memory-NIC data transfer.

Thus a typical RTP streaming transaction involves two memory-memory transfers and one memory-network transfer. Therefore, latency of an RTP streaming

transaction can be expressed as:

$$T_{RTP} = T_{CPU_{RTP}} + \frac{S}{32B_i} + \frac{4S}{B_i} + \frac{S}{B_e} \tag{3.9}$$

where $T_{CPU_{RTP}}$ is the CPU time taken by the RTP transaction response (in microseconds) and S is the size of PDU (in bytes).

3.4.2 HTTP Transaction Latency

A typical HTTP transaction at a Web server consists of two parts: request and response. Request part consists of a small HTTP command while the response consists of a header and the requested document. We can ignore the request part as its impact on overall transaction throughput is minimal. The rest of the transaction involves the following operations:

1. Preparation to send requested document in an HTTP response with a header. This response needs to be copied into a TCP buffer from HTTP. This process involves one memory-memory copy.

2. Calculation of TCP segment checksum requires entire segment to visit CPU (word-by-word, sequentially) through cache. Thus there is one memory-CPU data transfer for entire transport layer PDU.

3. Transfer of TCP segment to an IP buffer. This operation results in a memory-

memory transfer of entire PDU.

4. Finally, the IP packet is copied to the network buffer on the NIC resulting in a memory-NIC transfer.

HTTP transaction operations result in one memory-CPU transfer, two memory-memory transfers, and one memory-NIC transfer. Some CPU cycles are again needed for processing the request and forming response. As stated above, we can consider worst-case memory-CPU transfer with $MR = 1$ and latency is same as memory-memory transfer limited by the internal bus bandwidth. Using this approximation, total latency for an HTTP transaction that need highest throughput is given as:

$$T_{HTTP} = T_{CPU_{HTTP}} + \frac{S}{32B_i} + \frac{4S}{B_i} + \frac{S}{B_e} \tag{3.10}$$

where $T_{CPU_{HTTP}}$ is the CPU time taken by the HTTP transaction response in microseconds and S is the size of PDU in bytes.

3.4.3 IP Forwarding Latency

An IP packet forwarding transaction involves the following functions:

1. Copying incoming PDU to a buffer in the IP layer. This results in a NIC-memory transfer of entire PDU.

2. Examination of IP header to extract the destination IP address. This operation will result in copying IP header from memory to cache through a compulsory cache miss. Due to typically small size of the IP header (typically 20 bytes), it can completely fit in a cache line. A hash function is computed corresponding to the destination IP address to look-up the routing table to determine output port. This computation uses CPU cycles and can re-use the IP header from cache.

3. Routing table look-up involves an access to memory. However, over time, output ports corresponding to frequently encountered destination addresses will already be in cache. But in the worst case, table lookup will result in a cache miss and a memory-CPU transfer of typically one word.

4. IP header has to be updated such that time-to-live field is decremented and header checksum is recomputed. Since IP header is already in the cache, this function does not involve any additional latency.

5. Finally, the updated PDU with new header is transferred to the appropriate network interface (based on routing information) from IP layer. This results in a memory-NIC transfer of the entire PDU.

To summarize the entire transaction in terms of data transfers, there are two memory-NIC transfers of the entire PDU. In addition, there are two cache misses resulting in very small memory-CPU transfers. However, compared to memory-

memory transfers of entire PDUs, these memory-CPU transfers incur very small overhead of a few cycles only and can be ignored for all practical purposes. Thus, the total latency of each IP packet forwarding transaction can be given as:

$$T_{IP} = T_{CPU_{IP}} + \frac{2S}{B_e} \tag{3.11}$$

where $T_{CPU_{IP}}$ is the CPU time taken by the transaction in microseconds and S is the size of PDU in bytes.

Transaction latencies can be used to calculate the throughput (in MBytes/Sec) for a server running one of the three selected applications. Server throughput is given by $\frac{S}{T}$ where S is the size of transaction data and T is the latency of a transaction given by Equations 3.9, 3.10, and 3.11. In order to determine optimistic upper-bound on throughput for these three applications, we can apply two approximations to the latency expressions: (1) CPU usage latency compared to data transfer latency is negligible and can be ignored, and (2) bus contention from multiple simultaneously executed transactions do not result in any additional overhead. Then optimistic upper-bound on throughput for each application (in MBytes/Sec) is given as:

$$(Throughput)_{RTP} = \frac{32B_iB_e}{129B_e + 32B_i} \tag{3.12}$$

$$(Throughput)_{HTTP} = \frac{32B_iB_e}{129B_e + 32B_i} \tag{3.13}$$

$$(Throughput)_{IP} = \frac{B_e}{2} \tag{3.14}$$

We can use these upper-bound throughput estimates for several leading general-purpose microprocessors. These calculations are listed in Table 3.1. Using these calculations, we can conclude that all of the leading microprocessor based systems are capable of delivering more than 2 Gbytes/Sec throughput for all three applications. For high-end processors with high bandwidth internal system bus, the external bus becomes a major bottleneck in delivering high throughput. Despite this limitation, these upper-bound throughput estimates indicate that a general-purpose processor based server can deliver high throughput comparable to a server based on special-purpose architectures.

Table 3.1: Peak throughput of three network applications for leading general-purpose processors with different internal bus bandwidth. The external (e.g., PCI) bus is assumed to be 64 bits wide and operates at 133 MHz with a 1066 MBytes/Sec bandwidth.

Processor	Internal bus bandwidth (MB/Sec)	Throughput of three network applications		
		IP forwarding (Mbits/Sec)	HTTP (Mbits/Sec)	RTP Streaming (Mbits/Sec)
Intel Pentium IV 3.06 GHz	3200	4264	3640	3640
AMD Athlon XP 3000+	2700	4264	3291	3291
MIPS R16000 700 MHz	3200	4264	3640	3640
Sun Ultraspac III 900 MHz	1200	4264	1862	1862

Chapter 4

Measurement-Based Performance Evaluation

4.1 Introduction

In this chapter we present detailed measurement based memory performance evaluation of high throughput servers: streaming media servers, web servers and software routers. We begin by presenting our experimental testbed and measurement tools in Section 4.2. For each type of server, we present our choices of benchmarking tools, experimental factors, and performance metrics. Each of these servers executes on two operating system platforms: Linux and Windows. In order to focus our attention on the role of memory performance, we compare the performance of these two platforms in terms of memory bandwidth utilization, multithreading performance,

and context switching overheads in Section 4.3.

4.2 Experimental Testbed and Tools

Our experimental testbed comprises of a dual boot server machine that hosts one of the three high throughput servers: streaming media server, web server, and software router under one of two operating systems: Linux or Windows 2000 server. Six triple-boot machines will be used as clients to generate workload for the server.

Testbed

The setup consists of a closed-LAN with a Cisco 1 Gbps multilayer switch (catalyst 3550). Servers run on a PC with Pentium IV 2.0 GHz, 256 MB SDRAM, single 40 GB EIDE hard drive (Western Digital WD400) and 3Com 1 Gbps Ethernet NIC. The clients run on PCs, each comprising of a Pentium III 300 MHz, 96 MB RAM and 100 Mbps NIC. Figure 4.1 illustrates our experimental test bed.

The only difference in the testbed when we consider software routers is that we do not use the catalyst switch and the server has four NICs serving as router ports. Clients connect directly to the server through the ports. Linux operating system is configured to forward packets with *Routed* running as the routing deamon. *Routed* dynamically maintains a kernel routing table based on RIP (routing information protocol). Figure 4.2 illustrates testbed for software router.

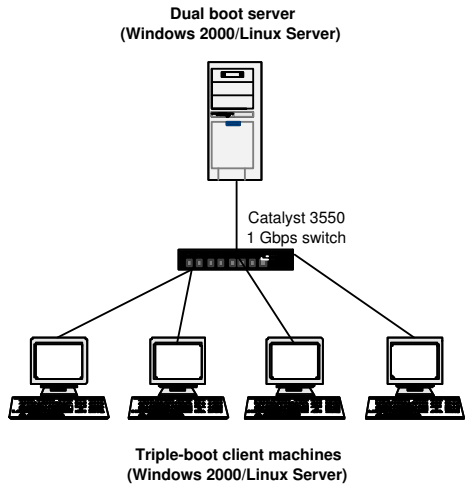


Figure 4.1: Experimental testbed consisting of a dual boot server and triple-boot client machines connected through Cisco catalyst 3550 switch.

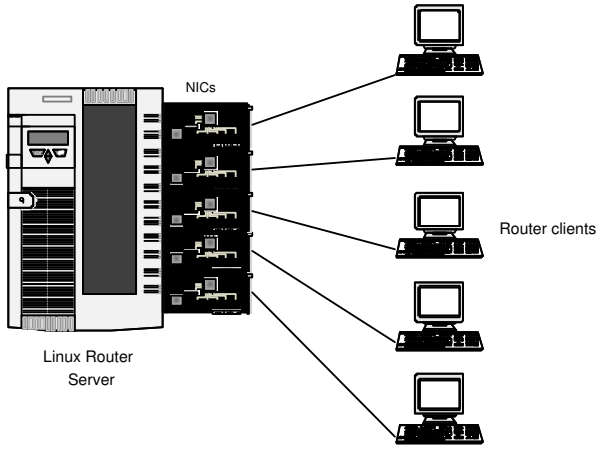


Figure 4.2: IP forwarding testbed consisting of router machine and routing clients.

Tools

We collect measurements for our metrics using a number of software tools that run on the server machine in a non-intrusive way. Some tools run on both platforms (Windows and Linux) while others run only on one platform. However, for platform specific tools, we ensure that such tools exhibit very similar overhead (generally minimally intrusive) on the specific operating system. The following tools were used to collect server performance statistics:

- VTune performance analyzer (6.1): VTune is an Intel tool for performance assessment and profiling of programs. VTune interfaces with Pentium processor on-chip performance counters. We collect performance data on both Windows and Linux using VTune.

- Windows 2000 performance monitor: This is a Windows platform performance tool. Performance monitor has features similar to VTune.

- *Netstat*: a tool for measuring bandwidth and observing network related activities. We monitor network connection status using *netstat*. It is available for both Windows and Linux.

- Linux tools: Some tools like *vmstat*, *iostat* and *sar*, are only available on Linux. We use them in place of Windows performance monitor since that is not available on Linux platform.

4.3 Analysis of Operating System Role

Our performance evaluation of three selected high throughput servers involve implementations on two platforms: Linux and Windows 2000 server. In this section, we investigate the role of these operating system on the performance of high throughput servers. We measure memory throughput on the two operating systems in our experiments. We also measure context switching overhead for the operating systems and compare the results. Our objective is to isolate any operating system level inefficiencies that may impact memory performance of high throughput servers.

4.3.1 Memory Throughput

To capture the impact of different operating systems, we conduct an experiment to analyze cache-to-memory and memory-memory throughput. ECT (extended copy transfer) *Memperf* [64] is a method to characterize the performance of memory systems. It captures two aspects of the memory hierarchy: its behavior with temporal locality by varying the working set size (block size) and the spatial locality by varying the access pattern (strides). Transfer bandwidth for a large volume of data is used as a metric. We conduct the extended copy transfer characterization for load sum test. The load sum test measures the memory throughput for all the block-sizes and access patterns.

Figure 4.3 shows our *Memperf* microbenchmark result for Linux and Windows

running on our server hardware. Both operating systems show similar memory performance, with the memory throughput decreasing as the block size increases beyond cache capacity. The worst case is when the block size is beyond 512 KB, which is the size of the level 2 cache. We run test for stride = 1, representing contiguous data. Based on these results, we conclude that difference in operating systems does not impact difference in memory performance of the servers. In both cases, a peak throughput of more than 5000 MB/Sec is observed, which is larger than Pentium IV system bus bandwidth of 3200 MB/Sec. Spatial locality due to multiple bytes of data blocks for each cache line results in high value of effective throughput.

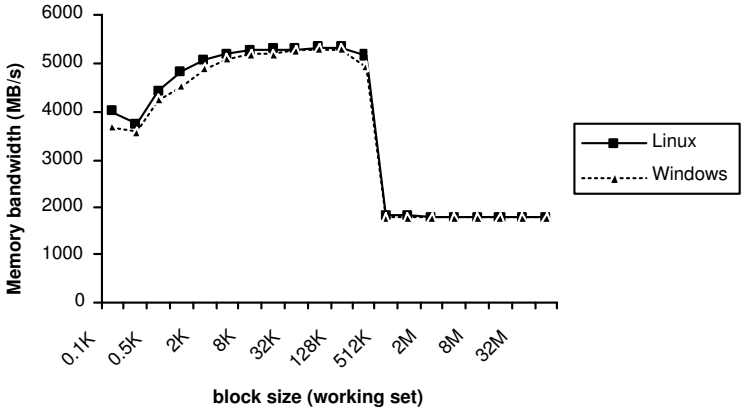


Figure 4.3: Extended copy transfer characterization (stride = 1).

4.3.2 Multithreading Support and Context Switching Overhead

Both Linux and Windows operating systems have multithreading support. Linux supports POSIX threads while Windows 2000 supports native windows threads. On multiprocessor systems, the operating system schedules different threads to execute on the different processors to improve performance.

We measure the overhead involved with context switching by running a simple test [65] that creates threads and pass a token back and forth between them for fixed number of times. The overhead of passing the token was shown to be negligible compared to the overhead of the context switch. The test program uses critical sections on Windows and pthread mutexes on Linux. Both Windows-critical sections and Pthread mutexes are considered locks. Initially, a lock is created for each thread. Each thread starts out owning a lock, which is a critical section on Windows and a mutex on Linux. Each thread locks its own lock and then attempts to acquire its neighbor's lock. When it acquires a lock from someone else, it releases the currently held lock and then attempts to get the next lock in sequence beyond the one it currently possesses. The sequential use of locks in this manner yields a trail of context switches which are measured. Figure 4.4 shows time per context for number of threads from 2 to 128. Windows 2000 clearly out-perform Linux as the time overhead for context switching in Linux is almost twice that in Windows.

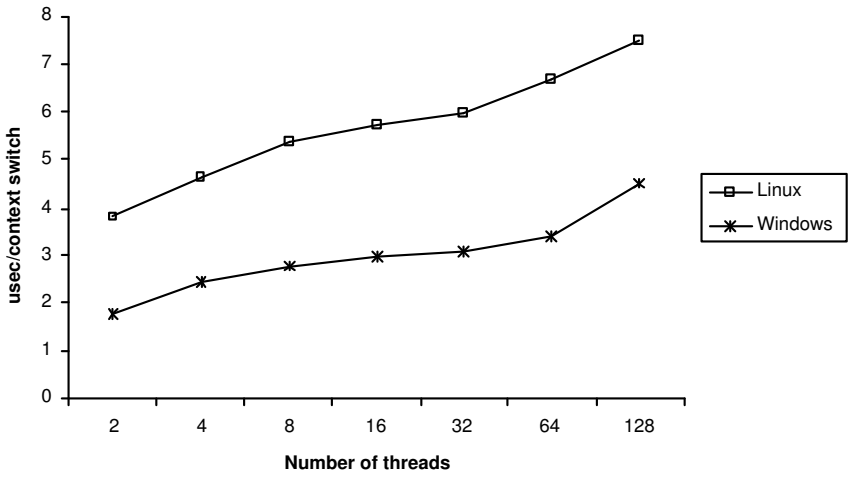


Figure 4.4: Context switching overhead on Linux and Windows.

4.4 Streaming Media Servers

Our experiment is basically video-on-demand scenario. Media clients make request for media objects. These objects are normally resident on disk in compressed form. The server starts streams to serve clients' requests. For protocols processing, the media object interacts with the processor in several ways. Protocol headers are appended, checksum is computed and finally passed to the network buffer. Processing an object involves memory-CPU copying, memory-memory copying, and memory-I/O copying involving copying a chunk from disk and when protocol processing is completed, the object is sent to the network interface.

We use two representative streaming servers: Apple Darwin streaming server and Microsoft media server. These servers are the most widely deployed on the Internet for streaming infrastructure. Another factor that lead us to use these servers is

because Darwin streaming server is available free while Windows media server ships with Windows 2000 server distribution.

4.4.1 Experimental Design

The experimental design is discussed in this section. We discuss the choice of factors based on our sensitivity analysis. The performance metrics are also presented.

Sensitivity Analysis

We conducted some initial experiments for our experimental design to determine the effect of factors and variation explained by each of the factors. Using $2^k r$ experimental design with replication, where $k = 3$ (number of client requests, encoding rate and stream distribution) and $r = 2$ (two replications), we computed the variation explained by each of these experimental factors. The number of client requests explains the highest variation (62.29% of total variation). Encoding rate (explained 19.33%) while stream distribution turns out to be marginally important (explained only 4.94%). All interactions of these factors explain negligible variation while experimental error explained a significant percentage (12.87%). High variation explained by experimental error could be attributed to random attributes in the load simulators, which make experiments not exactly repeatable.

Factors

We use experimental factors that enable us to observe the memory performance behavior of the servers. The following are the factors used for this experiment:

- Number of streams (streaming clients): We vary the number of media streams served. This corresponds to the number of media clients since each client is served by one stream. The range is from 1 client to 1000 clients.

- Media encoding rate (56 Kbps and 300 Kbps): We consider two extreme cases of media encoding rates: low end at 56 Kbps and high end at 300 Kbps.

- Stream distribution (unique or multiple media): Varying stream distribution between unique and multiple cases enables us to change the amount of data in the memory and also vary the rate at which disk will be accessed. Multiple stream distribution leads to a case where not all requested media can reside in memory, consequently leading to large disk access.

Metrics

We choose performance metrics that will enable us observe cache and memory subsystem behavior on the server. We also consider metrics like throughput and CPU utilization. The following is our list of metrics:

- Cache misses (L1 and L2 cache): On-chip caches are the fastest in the memory hierarchy. They however have very low capacity which necessitates frequent

accesses to higher units on the memory hierarchy. When the processor cannot find requested data in the caches, cache miss will result and the main memory will be accessed to obtain the data. Cache misses incur heavy penalty in terms of stall cycles. We measure cache misses using on-chip performance counters integrated into the processor. These are low level issues that require device drivers for interfacing.

- Page fault rate: Page fault results when the requested data is not available in memory. Page fault will lead to disk access. Disk access is very slow and the consequence is a memory access stall.

- Throughput: Throughput is another key index of performance of these servers. Servers are expected to deliver high throughput, especially when they are serving large number of clients.

- Server CPU utilization: This is the system wide non-idle CPU time.

4.4.2 Benchmarking Tools

To the best of our knowledge, there is no common benchmarking tool for streaming servers. This is due to the lack of one standard for implementation of servers. For instance both Darwin streaming server and Windows media server operate on entirely different protocols. We however ensured that our experiments are performed under the same conditions to deliver a fair assessment of the performance of the

servers.

We use streaming load tool to simulate streaming media clients. For Windows media server we use Microsoft media load simulator while for Darwin streaming server we use streaming load tool. Both simulators operate in similar manner, making requests for media objects through launching a large number of clients. This clients receive the requested media packet, examine it and discard after extracting required information. Since the clients do not engage in CPU intensive decoding and decompression of media data, they can support a large number of clients even on low end machines. We generate more than hundred clients on a Pentium III machine with 96 MB of memory.

4.4.3 Performance Evaluation

This subsection presents the results of our detailed measurements on streaming media servers. We discuss cache and memory performance. We also discuss the throughput and server CPU utilization.

Cache Performance

Figure 4.5 shows the L1 cache behavior under different configurations: number of clients, encoding rate, and stream distribution. Measurements for both Darwin Streaming Server (DSS) and Windows Media Server (WMS) are reported. Both Figures (a) and (b) show increase in cache misses as the number of clients increases.

Thought not to a large extend, the stream distribution and encoding rate also affect the miss rate. When a client request for a media content, protocol processing overhead is incurred, leading to multiple memory-to-memory copying and memory-to-network copying operations. Though in the case of unique distribution, data may not always be fetched from disk for all clients, protocol processing is performed separately for each stream. These protocol processing overheads make the number of clients the main contributor to the large number of cache misses at both level 1 and 2 caches.

We observe the worst case cache misses in both L1 and L2 when there is a large number of clients requesting multiple streams at 300 Kbps encoding rate. For this case, we started observing clients being refused connection by the server, which eventually makes a client to time out. We show L2 cache behavior for 56 Kbps and 300 Kbps encoding in Figure 4.6. For our hardware: Pentium IV processor, L1 cache is a data cache only while L2 is a mixed cache. This is why we observe a higher number of L2 cache misses compared to L1 cache. For all these cases, Windows media server exhibits lower L1 and L2 cache misses. This is most probably due to cache-friendly design of WMS on Windows 2000 platform.

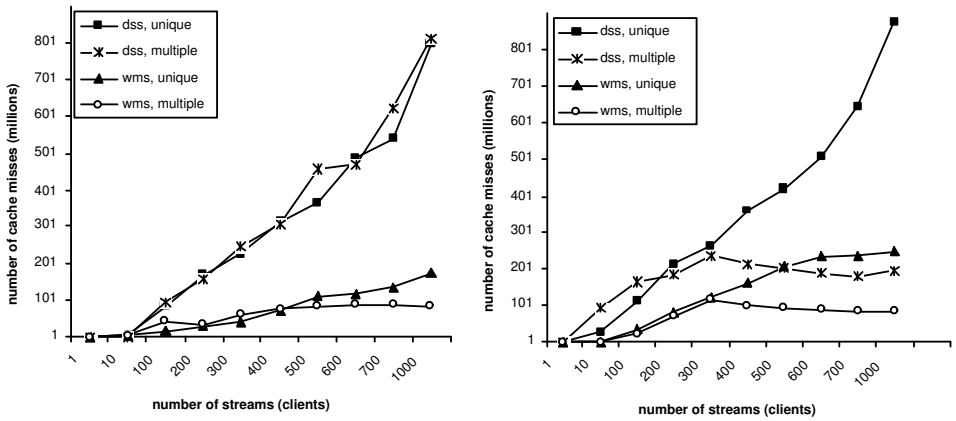


Figure 4.5: Server L1 cache misses: (a) at 56 Kbps encoding (b) at 300 Kbps encoding.

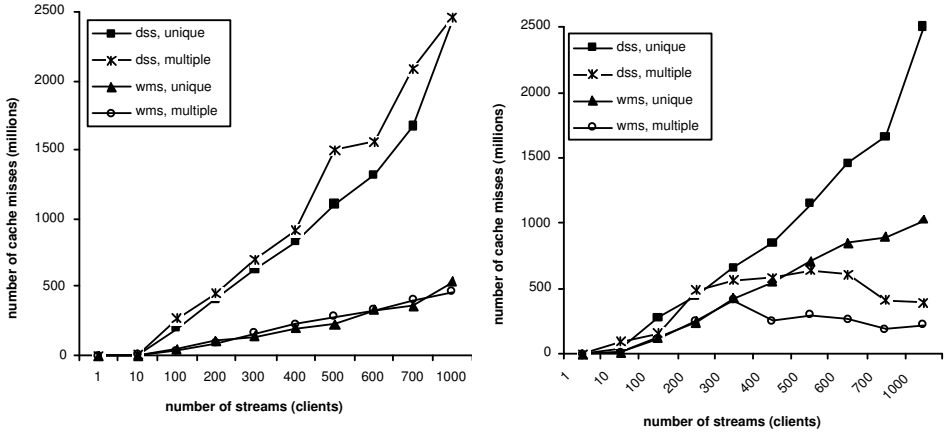


Figure 4.6: Server L2 cache misses: (a) at 56 Kbps encoding (b) at 300 Kbps encoding.

Memory Performance

Main memory performance is considered in terms of page fault rate. Any data referenced in the memory that is not available will be fetched from the disk. This is an expensive and slow process. Disk access is very slow especially when frequent disk references occur within a short duration. Disk is an I/O device attached to the I/O bus with access time greater than that for RAM. Since data chunks may not be necessarily stored in contiguous location on the disk, random seek time to access some data can significantly increase the bottleneck. When all clients make requests for the same media content (unique distribution), disk access is highly minimized since the objects are likely to be served from memory for subsequent accesses. However, when the stream distribution is multiple, several objects are fetched from disk to memory. The memory cannot accommodate all objects, hence served objects are flushed from memory. This means any future request for the same object that was previously removed from memory will involve disk access again. This situation leads to significant degradation in performance. It also results in more memory copying since any data fetched from the disk will have to be buffered in memory.

Figure 4.7 shows the page fault rates. When the distribution is unique, most memory reference will hit in the TLB and there are fewer page faults. This is shown by an almost flat page fault rate across all number of clients and this is same for

all encoding rates. However, for multiple distribution, the page fault rate increases with increase in number of clients, indicating more disk activity.

The consequence of high page fault rate shows in clients' timeout. It is obvious that as more disk activity is involved, disk access latency is exacerbated and the server responds to clients' request very slowly. Clients have time limits, after which they timeout. We observe a high number of clients' time out when there is a large number of clients requesting large number of different media objects at high encoding rate.

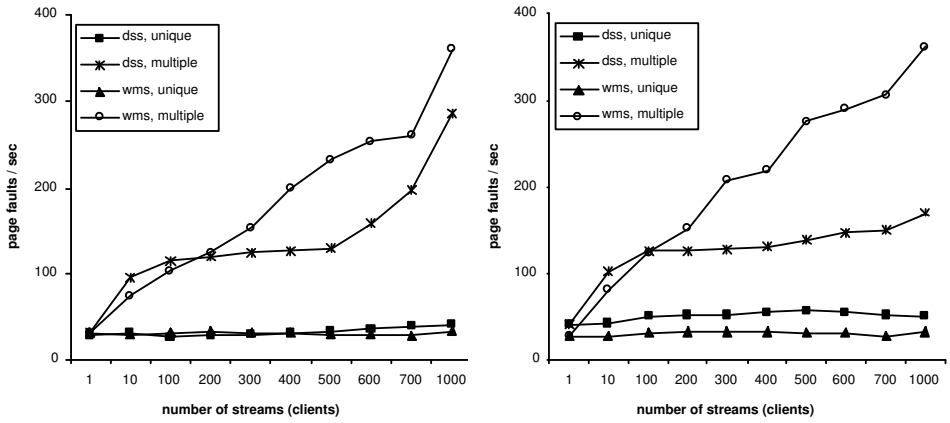


Figure 4.7: Server page fault rates (a) at 56 Kbps encoding (b) at 300 Kbps encoding.

Throughput and CPU Utilization

We measure throughput as the total bytes delivered by the server per second. Figure 4.8 shows the throughput for 56 Kbps and 300 Kbps encoding rates. When the number of clients is large, the server delivers at a higher bytes per second rate. This

is especially true for low encoding rate where we do not experience clients timeout. Since each client is served by a stream, the aggregate throughput for large number of clients is high. This is obvious from the figure when the encoding rate is low. However as we explained in the previous section, when clients are requesting multiple media objects at high encoding rate, the number of clients timing out increases significantly. We reach a situation where the number of clients is effectively less, resulting in reduction in throughput. This is particularly obvious in Figure 4.8(b). It corresponds to the situation where number of clients is beyond 100 and clients are requesting multiple 300 Kbps media objects. We report higher throughput for Windows media server for all the cases. Especially using multiple streams, WMS shows higher throughput than DSS despite a greater page fault rate. It means that WMS does a superior job hiding the latency of these page faults with useful streaming media transaction processing.

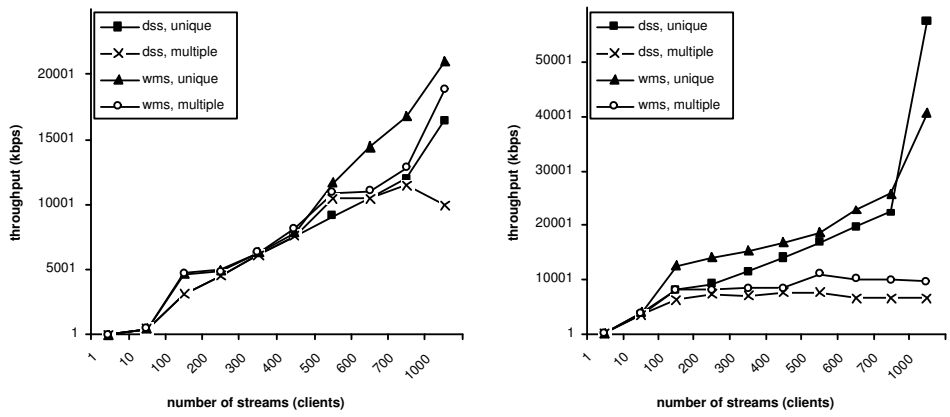


Figure 4.8: Server Throughput (a) at 56 Kbps encoding (b) at 300 Kbps encoding.

The CPU utilization is shown in Figure 4.9. There is increase in CPU utilization as the number of clients increases. For unique stream access, the CPU utilization increases all the way up to 1000 streams. CPU utilization begins to fall when the number of aborted clients due to time outs increases, as we explained earlier.

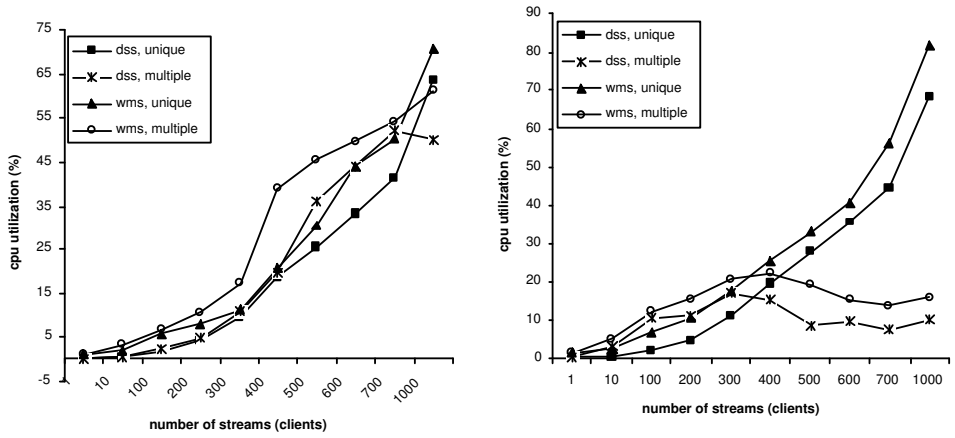


Figure 4.9: Server CPU utilization (a) at 56 Kbps encoding (b) at 300 Kbps encoding.

Table 4.1 summarizes our findings on memory performance evaluation of streaming media servers. The table presents these findings while comparing the performance of DSS and WMS. This comparison shows the importance of memory performance tuning to obtain high server throughput. As we observe in WMS case, simple latency hiding techniques, such as multithreading, can ensure higher throughput even when excessive disk usage results in higher page fault rates.

Table 4.1: Comparison of streaming media server performance with respect to selected metrics.

Metrics	**Applications**	
	Darwin Streaming Server	**Windows Media Server**
Cache miss	Highest degradation in cache performance (both L1 and L2) when the number of clients is large and the encoding rate is 300 Kbps with multiple multimedia objects.	Exhibits same cache behavior as Darwin streaming server but incurs fewer cache misses for both L1 and L2.
Page faults	When clients demand unique media objects, page fault rate is constant. However, if the request is for multiple objects, the page fault rate increases with the number of clients.	The same characteristics as with Darwin streaming server. It incurs more page faults at multiple distribution requests.
Throughput	Throughput increases with number of clients. Higher encoding rate – 300 Kbps, also accounts for more throughputs. Darwin streaming server has less throughput compared to Windows media server.	WMS shows same or higher throughputs compared to DSS under identical operating conditions, the bandwidth characteristics are same with Darwin streaming server.
CPU utilization	Higher CPU utilization, especially with unique streams. This is consistent with latency hiding observation.	Same CPU behavior pattern, but has higher CPU utilization.

4.5 Web Servers

A typical HTTP transaction consists of a request and a response. Request from a client consist of a small packet carrying an HTTP command, such as GET. The response comprises of an HTTP header followed by the requested object. When the server receives a HTTP request, it invokes required protocol processing to extract the information from the received packet header. The server responds by fetching the requested object, likely from a disk to the main memory, passing it for processing that involves memory-CPU copying, memory-memory copying, and memory-I/O copying. Finally the object is forwarded via the network protocol stack to the network buffer for delivery to the client.

Two representative web servers: Apache web server and Microsoft Internet Information server (IIS) are used. These servers are the most widely deployed on the Internet for WWW applications. Another factor that lead us to use these servers is because Apache web server is available free while IIS ships with Windows 2000 server distribution.

4.5.1 Experimental Design

The experimental design is discussed in this section. We discuss the choice of factors and metrics for these experiments.

Factors

Experimental factors that enable us to observe the memory performance behavior of the servers are used. The following are the factors used for this experiment:

- Number of WWW clients: This is the number of clients requesting HTTP documents from the server. Each client sends a HTTP request for a particular web document while the server responds to the request as fast as it could. All HTTP requests are for static documents. We vary number of clients from 1 to 400.

- Document size: We vary document size from very small size (5 bytes) to very large size (50 Mbytes). The set of our document represents a wide range of document size distribution popular on the Internet, especially the range 10 KB to 50 KB.

Metrics

Performance metrics that will enable us observe cache and memory subsystem behavior of the servers are chosen. We also consider metrics like throughput and CPU utilization. The following is our list of metrics:

- Cache misses (L1 and L2 cache): On-chip caches are the fastest in the memory hierarchy. They however have very low capacity which necessitates frequent accesses to higher units on the memory hierarchy. When the processor cannot

find requested data in the cache, cache miss will result and the main memory will be accessed to obtain the data. Cache misses incur heavy penalty in terms of stall cycles. We measure cache misses using on-chip performance counters integrated into the processor. These are low level issues that require device drivers for interfacing.

- Page fault rate: Page fault results when the requested data is not available in memory. Page fault will lead to disk access. Disk access is very slow and the consequence is a memory access stall.

- Throughput: Throughput is another key index of performance of these servers. Servers are expected to deliver high throughput, especially when they are serving large number of clients.

- Server CPU utilization: This is the system wide non-idle CPU time.

- Transactions/Sec (connection rate): It is a measure of how fast the web server receives client requests and responds to such requests.

- Average latency: It is the latency observed by a client from the time the client sends a request until it receives a response. Large latency means poor server performance. Sometimes large latency might be as a result of congestion in the network.

4.5.2 Benchmarking Tools

We use popular web benchmarking tool known as *Webstone* [66]. *Webstone* was used to generate large HTTP requests to the web servers. *Webstone* is a configurable benchmark tool that allows performance measurement of web servers. *Webstone* was originally developed by Silicon Graphics. *Webstone* 2.5 is Mindcraft's enhancement to *Webstone* 2.0.1 to improve reliability and portability as well as to make tests more reproducible. *Webstone* creates a load on a web server by simulating the activity of multiple clients, which are called web clients and which can be thought of as users, web browsers, or other software that retrieves files from a web server. In other to create large loads on a web server, *Webstone* is able to distribute WWW clients among client computers. The Webmaster is the program that controls all of the testing done by *Webstone*. With *Webstone*, we can measure average and maximum connect time (delay), average connection rate, average and maximum response time and data throughput rate.

4.5.3 Performance Evaluation

This section presents the result of our detailed measurements on web servers. We discuss cache and memory performance. Our primary objective is to analyze the impact of memory performance on web server throughput. We present our finding by comparing the performance of two popular web servers: Apache and Microsoft IIS.

Web Transactions

A web transaction is the series of client-server interactions that include: (1) a client establishing a TCP connection with the server; (2) client sending its HTTP request; (3) the server responding with the required document if available or with an error code; and (4) the termination of the TCP connection. To make our discussion clearer, we start by considering the relationship between web document size and number of transactions. The size of the document requested by a web client is highly significant for the performance of the web server in terms of both hardware and software. Figure 4.10 shows that as the document size increases, the number of transactions becomes smaller. For a large document size, the server and client must maintain a connection for a longer time to transfer the file to the client. In this case, connections are established and terminated over a relatively longer time compared to when the document is small. Establishing a connection and terminating it frequently will incur a heavy performance cost on the server side, especially when there is a large volume of such transactions. As shown in the Figure 4.10, when the document size is small, the rate of transaction is high. When the client requests a small document, the server can send the entire document in one packet and tear down the connection. As soon as the client receives the document, it sends another request. On the other hand, when the requested document is very large, the server sends the document as chunks in several packets. As long as the client is still

receiving a response for a previous request, it cannot establish another connection, resulting in low connection rate. Generally, the two servers perform poorly for very small web documents. This observation is further supported by another study on web servers by Hu et al [56]. Compared to Apache, IIS exhibits higher transaction throughput for small to moderately large file sizes.

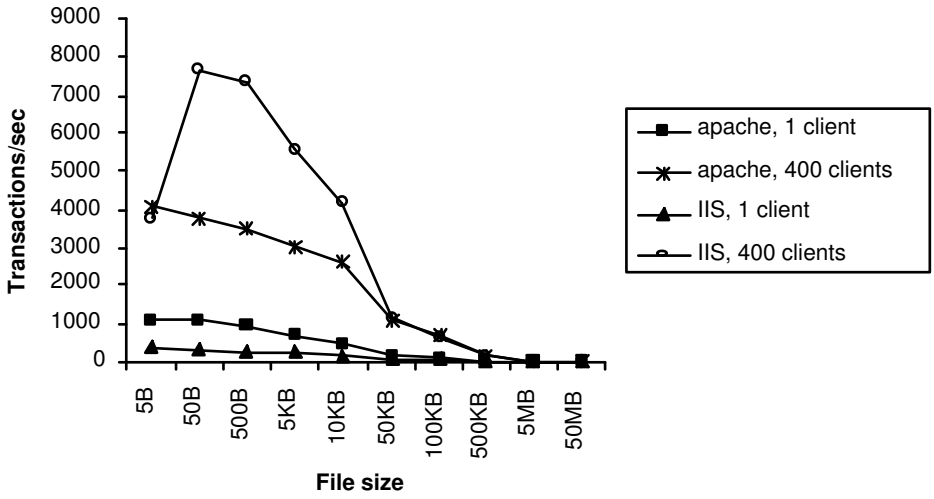
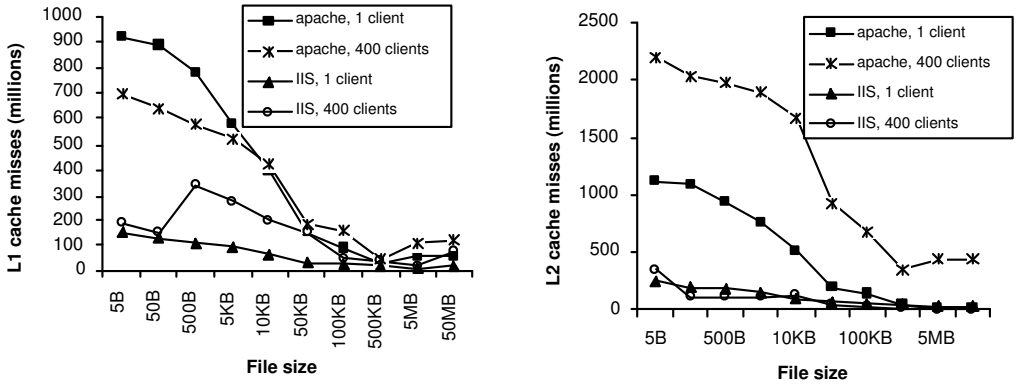


Figure 4.10: Variation of server transactions with file size.

Cache Performance

As shown in Figures 4.11(a) and (b) , both L1 and L2 caches perform poorly for small documents. Although this is surprising, it follows from the effect of document size on the number of transactions as mentioned earlier: if the document size is small, more connections are established and released within a short time. Frequent connection establishment and release results in more activity for the processor and

large number of cache misses (as will be shown in CPU utilization plot in the next section). Apache performs worst in terms of cache misses. Apache is a process-based server, which forks several processes that serially accept new connections. Although Apache server tries to minimize the overhead of forking new processes by pre-forking a pool of processes at initialization, the server resorts to forking a new process for every request during heavy loads. New process creation is a CPU intensive activity and leads to excessive cache misses. Generally, the two servers have poor cache performance for small documents.



(a) L1 cache misses

(b) L2 cache misses

Figure 4.11: Variation of server cache misses with file size.

Memory Performance

Memory performance in web servers is similar to streaming servers. A large document cannot completely reside in memory and can only be served in smaller chunks. Therefore, a large document will have to be continuously served from the disk using

virtual memory subsystem. Every access to the disk is likely to incur a page fault and a disk I/O operation. As was shown in Figure 4.7, high page fault rate was observed when the document size is large. While a small document is likely to be in memory after a recent access, the large documents must be read from the disk. Such transfer from disk to memory involves moving data over the I/O bus to internal bus and finally copying them to main memory. With large number of such transfers, performance penalty becomes significant. The page fault rate shown in Figure 4.12 indicates that the page fault rate increases as the document size increases. When the document size is 50 MB, the page fault rate is too high, resulting in long delays to serve clients' requests. At this stage, we observe frequent client time outs. We observe that the web server performs optimally with an acceptable latency and high bandwidth, when the document size is neither too small nor too large (see Figure 4.13).

Figure 4.12 also indicates that Apache web server incurs larger page fault rate compared to IIS when file size is unusually large in the range of 50 MB. I/O and network interfaces under Linux appear to favor larger file transfers more compared to small and medium length files. This observation can again be confirmed by considering throughput and CPU utilization plots (See Figure 4.13).

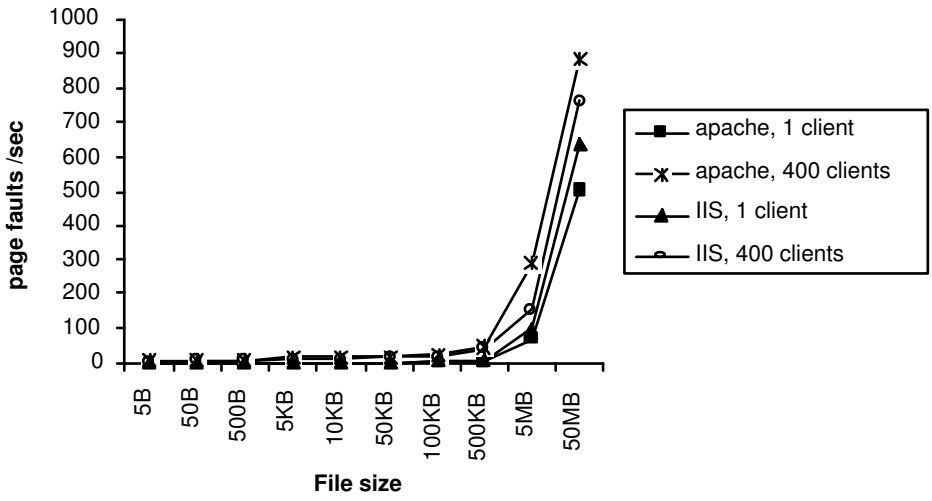
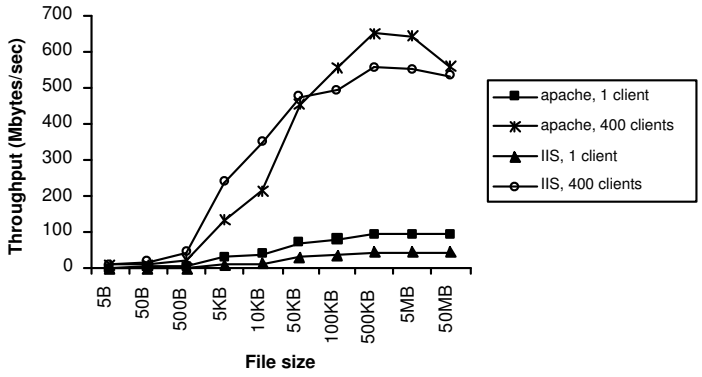


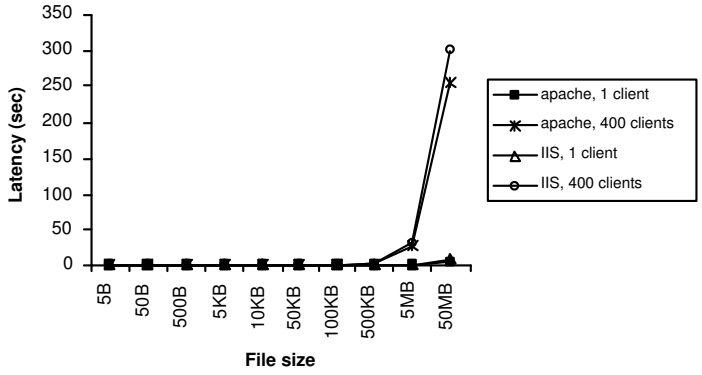
Figure 4.12: Variation of server page fault rate with file size.

Throughput, Latency, and CPU Utilization

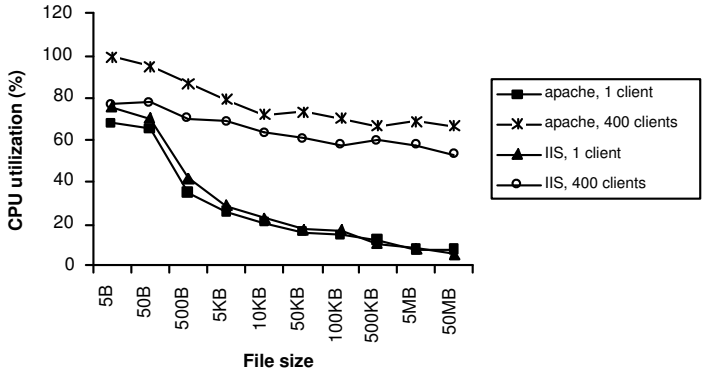
Peak throughput was observed at 500 KB and 5 MB document sizes as shown in Figure 4.13(a). However, at this point of high throughput, the latency is very high (see Figure 4.13(b)). If clients experience long latencies in accessing documents from a web site, such clients may abandon the request and switch to other web sites or may not visit the site again. So it is very important for web servers to serve clients' request as quickly as possible. For all cases, Apache server delivers higher throughput compared to IIS. Generally, throughput increases with document size.



(a) Throughput



(b) Latency



(c) CPU utilization

Figure 4.13: Variation of server (a) throughput, (b) latency and (c) CPU utilization with file size.

For small document sizes, clients generally experience low latency. If the document is large, in the range of 5 MB to 50 MB, the latency becomes too high and clients might even timeout. Server latency is shown in Figure 4.13(b). A major contributor to server latency is disk access time. Our server hardware has a single disk, which can easily become a bottleneck. Other factors contributing to high latency are cache misses and page faults because of their resulting high penalty in terms of CPU cycles.

CPU utilization is shown in Figure 4.13(c). It is easy to saturate the server (100% CPU utilization) when the requested document is small. As we explained earlier, when the requested document is small, the number of transactions per second (connection rate) becomes high and more connections are setup and terminated resulting in more CPU utilization.

While evaluating web server performance, we should not overlook the fact that most of the HTTP transactions transfer objects that are close to 10 KB in size. Table 4.2 compares Apache and IIS web servers at this critical file size. IIS shows about 58% higher throughput over Apache. The only notable difference between the performance of the two is remarkably lower L1 and L2 cache misses for IIS at this critical operating case.

Table 4.2: Comparison of Apache and IIS web servers for an average file size of 10 KB.

Attribute	Value	
	Apache	IIS
Max. transaction rate (Conn/Sec)	2586	4178 (58% more than Apache)
Max. throughput (Mbps)	217	349 (62% more than Apache)
CPU utilization (%)	71	63
L1 misses (Millions)	424	200
L2 misses (Millions)	1673	117
Page fault rate (PFS/Sec)	< 10	< 10

Table 4.3 summarizes our discussion on memory performance evaluation of two popular web servers: Apache and Microsoft IIS.

Web server performance evaluation again emphasizes the critical role of memory subsystem performance in delivering high throughput. At a critical file size of 10 KB, IIS shows more than 50% higher throughput (both in terms of number of transactions per second and total bytes transferred per second) compared to Apache due to superior cache performance. Considering all else being identical in Linux and Windows 2000 server, IIS protocol processing is more cache friendly on Windows platform compared to the same for Apache on Linux, resulting in significantly high throughput.

Table 4.3: Comparison of Apache and IIS web server performance.

Metrics	**Applications**	
	Apache Web Server	**Internet Information Server**
Cache miss	Highest cache misses for both L1 and L2 are observed when the document size is small. Minimal number of cache misses is recorded for the largest document (50 MB).	The same pattern of cache misses with Apache server. IIS has a much better cache performance (lower cache misses) compared to Apache server.
Page faults	Very low page faults when the document size is small. However, at very large document size, the page fault rate is very high.	Same behavior with Apache server.
Throughput in Mbps	Higher bandwidth is recorded for larger number of clients requesting document of large size. Apache recorded a highest bandwidth of 649.43 MB/Sec.	Highest bandwidth recorded is 553 MB/Sec.
Transactions per second	Number of transactions per second decreases as the document size becomes larger. The highest transaction was recorded at 4064 per second.	With similar pattern, IIS recorded a higher number of transactions at 7633 per second.
Average latency	Clients perceive the highest latency when the document size is very large.	Clients perceive the highest latency when the document size is very large.
CPU utilization	Higher CPU utilization is observed for smaller documents. Apache has higher CPU utilization.	IIS has less CPU utilization compared to Apache. Both show the same pattern in CPU activity

4.6 Software Router

Software routers are implemented on general-purpose PC platforms. In addition to routing and forwarding of IP packets, these routers can utilize the general purpose platform to provide other value added services, including differentiated services, packet filtering, firewalls, and load balancing. Since packet forwarding is the most frequent of all of these services, we will focus on performance evaluation of software router with respect to it.

In IP forwarding, the packet is intercepted from the NIC and copied to the IP layer, which results in NIC-memory transfer of the entire packet. To examine the packet header in the IP layer, it is copied to the processor cache resulting in a compulsory cache miss. The router forwards the packet by examining the routing table, which also involves memory access. Updating the IP packet header also involves memory access. The packet is finally forwarded to the selected interface. In the case of high throughput IP forwarding on software routers involving multiple NICs, bus contention becomes a significant issue that can inhibit performance. Even though the bus bandwidth may be high, performance is likely to be marred due to bus contention and context switching of processes that are responsible for routing and forwarding.

4.6.1 Communication Configurations

Since routers determine communication path between clients, different communication paths are explored with the aim of observing performance for different communication configurations. We are able to evaluate the performance of IP forwarding for several instances of client-to-client communication route. Figure 4.14 illustrates the communication configurations.

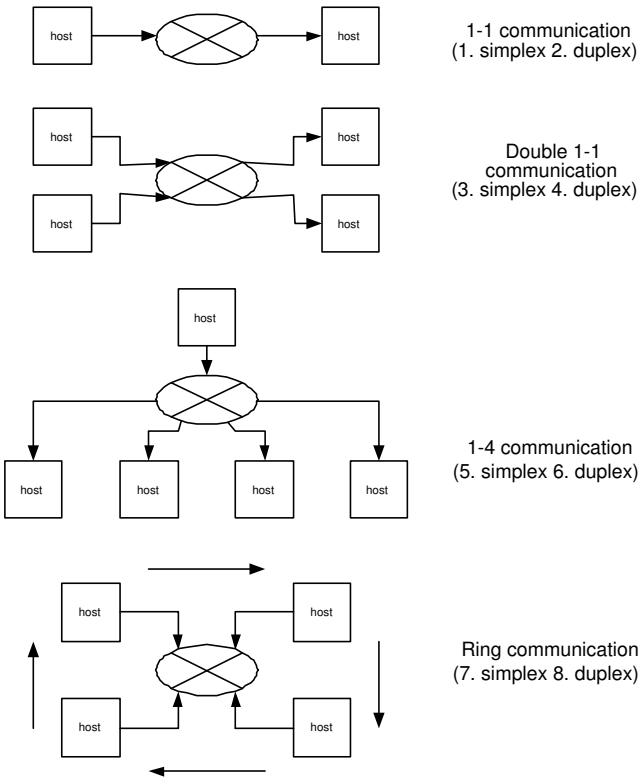


Figure 4.14: Routing configurations showing simplex and duplex modes.

4.6.2 Experimental Design

The experimental design is discussed in this section. We discuss the choice of factors and metrics for these experiments.

Factors

We use experimental factors that enable us to observe the memory performance behavior of the servers. The following are the factors used for these experiments:

- Routing configurations in eight levels as shown in Figure 4.14.
- TCP message size: We observe the effect of IP payload size by varying the TCP message size. We consider small packet size at 64 B, medium size at 10 KB and large size at 64 KB.

Metrics

We choose performance metrics that will enable us observe cache and memory subsystem behavior on the server. We also consider metrics like throughput and CPU utilization. The following is our list of metrics:

- Throughput: Throughput is another key index of performance of these servers. Servers are expected to deliver high throughput, especially when they are serving large number of clients.
- Server CPU utilization: This is the system wide non-idle CPU time.

- Number of context switching: System context switching for high performance IP forwarding is a source of significant overhead. We measure number of context switching per second.

- Number of active pages: Recently touched pages (normally 4 KB per page in Linux) in memory. Number of active pages provides insight on memory activity of the software router.

4.6.3 Benchmarking Tools

Industry standard benchmarking tool for networks, *Netperf* [67] is used. *Netperf* is a network performance benchmarking tool that can be used to measure various aspects of network performance. *Netperf* can generate high network traffic for both TCP and UDP, and was used to generate IP traffic for the software router. For these experiments, *Netperf* was used to generate IP traffic (TCP) from the clients, which have to be routed based on the destination IP address.

4.6.4 Performance Evaluation

Our major concern is IP forwarding at the highest throughput possible while also examining system activities like CPU utilization and context switching. We focus on memory by observing the variation of active memory pages – how frequently memory pages change. We report measurements on interface throughput, context

switching per second, CPU utilization and number of active pages per second.

Throughput

A peak throughput of 449 Mbps is observed. This is less than our interface capacity of 1 Gbps. This bandwidth is achieved for configuration number two – 1-to-1 communication in duplex mode. In this configuration, only two interfaces (NICs) on the forwarding machine are involved in the routing. With only two NICs involved, bus contention is minimal and the context switching is also low. Throughput measurements for all the interfaces are shown in Figure 4.15.

Configuration number six (full duplex 1-to-4 tree communication pattern) also indicates high throughput through interface eth0 on the server. In this configuration, one host sends packets to three other hosts through eth0, hence the interface eth0 carries the aggregate traffic for three machines, leading to high traffic flow. In general, the throughput of a software router for small packets is low. When the packet is only 64 B, a lot of overhead is incurred as large number of context switching is observed. For instance, the IP header is 20 B minimum and the TCP header is also 20 B, which means an overhead of more than 60%. Regardless of the packet, it has to go through the same protocol processing, resulting in low byte payload, which inhibits high throughput IP packet forwarding.

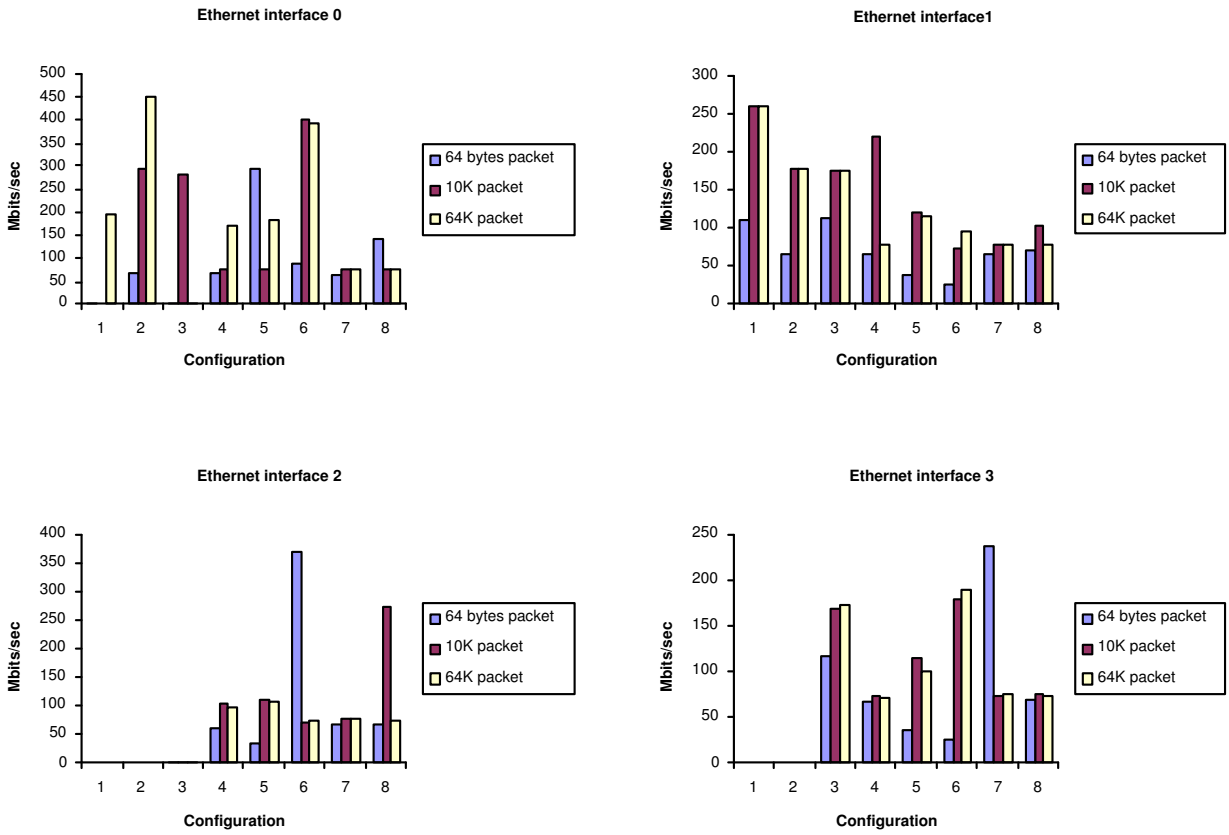


Figure 4.15: Router NICs throughput.

Context Switching

Number of context switching per second indicates a measure of how the system switches between NICs to use the shared I/O bus . We report the highest context switching rate of 5378 per second, which was observed for configuration number eight (full duplex ring communication pattern) when the packet is small (64 B). In this configuration, all NICs are involved in passing traffic in both directions. We expect the highest bus contention in this case, resulting in low overall traffic throughput. The context switching rates for all eight configurations are presented in Figure 4.16.

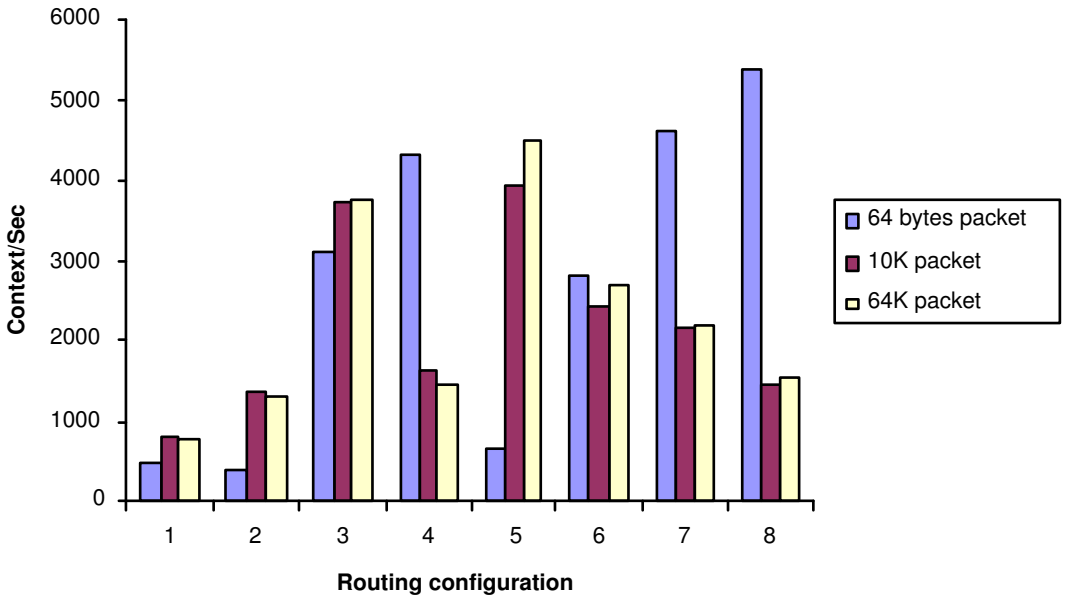


Figure 4.16: Variation of context switching rate with routing configuration.

Active Page

As shown in Figure 4.17, the number of active pages is almost uniform throughout the eight routing configurations. It does not show significant variation with packet size as well. This is an indication that memory activity is not very high. Generally, packet payload is not modified during forwarding. Only the IP header undergoes modification to update, for instance time-to-live (TTL) and header checksum fields. But since the header is very small, this modification does not trigger large memory activity.

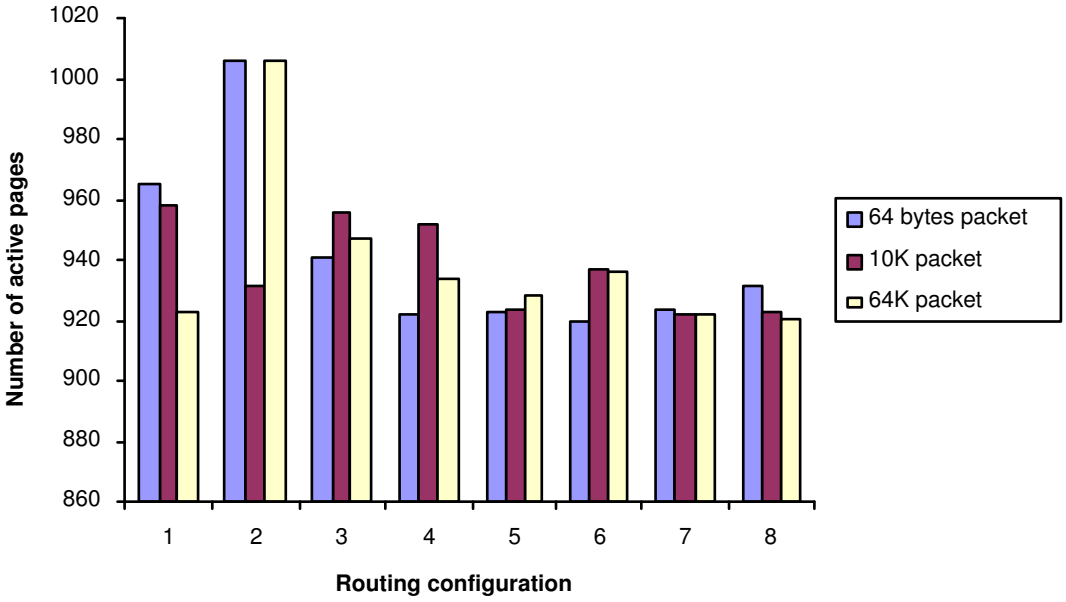


Figure 4.17: Variation of number of active pages with routing configuration.

CPU Utilization

All CPU activity was at the kernel level. In fact, we observe almost zero CPU utilization in the user space. The highest CPU utilization of 84% was observed for configurations four and eight when the packet size is medium and large. These configurations both involved full duplex communication, utilizing all interfaces. The lowest CPU activity on the other hand is for configuration one. These measurements clearly show that as we have more interfaces involved in forwarding, the CPU utilization also increases. This is intuitive as we observe more context switching for these configurations. CPU utilization is shown in Figure 4.18.

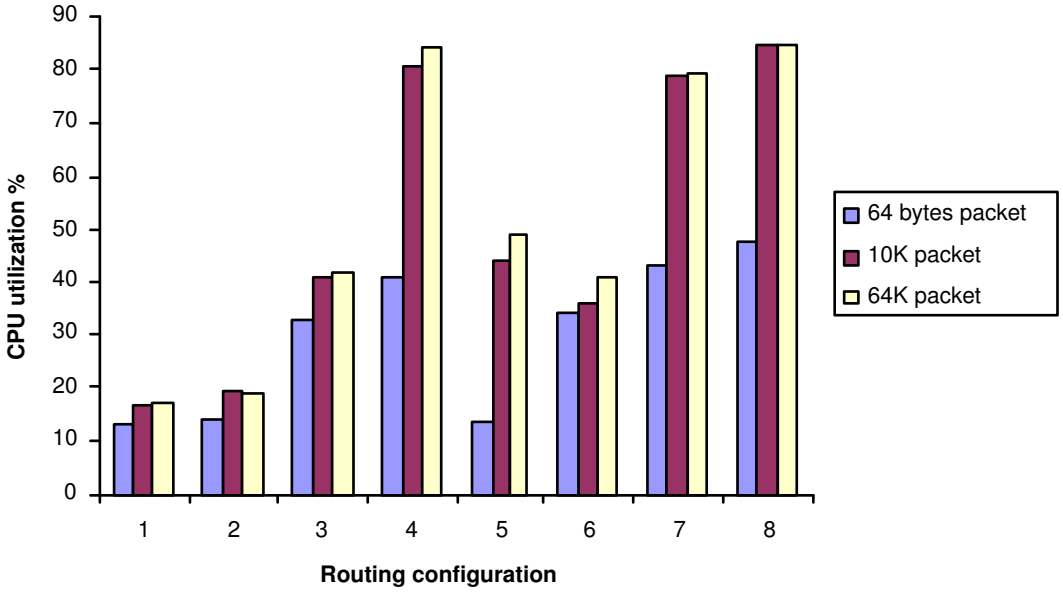


Figure 4.18: Variation of CPU Utilization with routing configuration.

For high performance IP forwarding, main performance constraints are likely to be CPU and bus contention, which manifests in the form of large number of context switches. It is obvious that as we increase the number of interfaces on the routing server, we would hit CPU saturation for most routing configurations and the number of context switches is likely to increase.

Table 4.4 summarizes our discussion on memory performance evaluation of software router.

Table 4.4: Summary of IP forwarding performance.

Metrics	**Performance**
Throughput	IP forwarding attained a maximum bandwidth of 449 Mbps for configuration number 2 – full duplex one-to-one communication.
CPU utilization	The highest CPU utilization of 84% was observed for configurations four and eight when the packet size is medium and large. These configurations both involved full duplex communication utilizing all interfaces.
Context switching	Highest context switching rate of 5378/Sec was observed when more interfaces (NICs) are involved in forwarding packets, indicating higher contention for shared I/O bus.
Number of active pages	Fairly uniformly distributed active page figures indicates that memory activity is not very intensive.

4.7 Summary

We discussed memory performance evaluation for representative high throughput servers. Our study focused on memory performance issues including L1 and L2 cache misses, page fault rates, and other metrics directly related to server throughput. For streaming media servers and web servers, cache misses and page fault rates are significant when the number of clients is large, and these are sources of performance limitation. What we observed as main performance limiting factors for software routers are context switching overheads and CPU utilization. As we have more ports on the router, context switching will increase significantly and CPU will be saturated.

The highest throughputs measured are: 57.36 Mbps, 649.43 Mbps and 449 Mbps for streaming server, web server and software router, respectively. In all these cases, the measured throughputs are much less than the calculated peak in Chapter 3. Though our peak throughput estimates were based on simplifying assumptions, it is still possible to obtain throughputs higher than what we measured if we fine tune the operating systems and enhance the design of the servers to hide latency.

Chapter 5

Design, Implementation, and Performance Evaluation of a Double Buffer RTP(DB-RTP) Server

5.1 Introduction

In this chapter, we discuss the implementation of a prototype RTP server designed to hide disk access latency by pre-fetching and read-ahead buffering. Among the three types of high throughput servers we are considering in this thesis, our implementation choice of streaming server is due the non-availability of an open standard

based implementation of the streaming servers incorporating techniques to enhance memory and/or disk performance. The design utilizes double buffer in the memory to simplify buffer read/write synchronization . Due to sequential nature of media stream access, blocks that are required later for playback can be pre-fetched ahead of time and stored in the buffer. Read ahead buffering increases the deadline bound of a request, leading to a decrease in the loss probability [43] and will minimize jitter. We also present a detailed measurement-based performance evaluation of DB-RTP server.

5.2 Design Overview

In this section, we outline the architecture of our DB-RTP server. We discuss the design concepts of pre-fetching and double-buffering as well as the synchronization mechanism.

5.2.1 Architecture

Figure 5.1 illustrates the architecture of our double buffer RTP server. In this design, a pair of memory buffers is available for each stream. The server streams to the client from these buffers, alternating between the two buffers. Media object chunk is fetched ahead of time and buffered into one of the memory buffers. The RTP packetizer retrieves the media chunk, appends the RTP header, and passes it

to the UDP/IP protocol stack. To respond to clients' request, we have implemented a stream-lined RTSP server, which listens on the standard RTSP port number 554. All clients direct their requests for media objects to the RTSP server. The RTSP server parses the request and schedules a new stream to fulfill such requests.

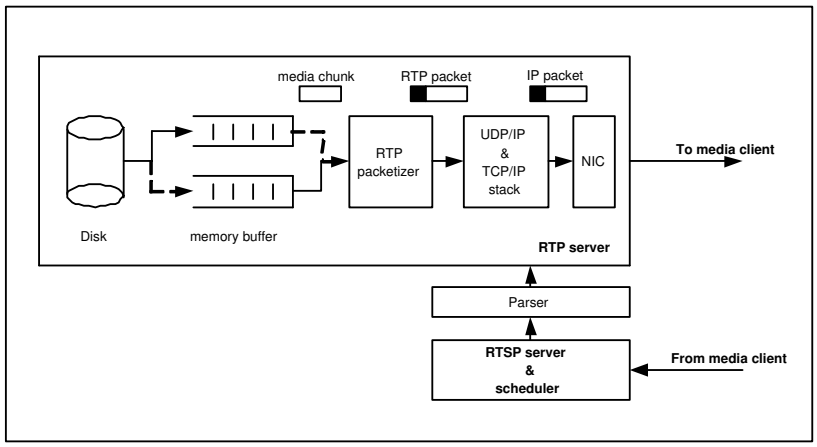


Figure 5.1: Architecture of a double buffer RTP server.

5.2.2 Double Buffering and Synchronization

We employ double buffering to simplify synchronization procedure when accessing and writing to the buffers. While one buffer is refilled, the other buffer will be read. Synchronization is explicit as we describe shortly. The buffering procedure is shown in Figure 5.2.

When the RTSP server schedules a client's request, a block of the media chunks is fetched from the disk and written to buffer A. This block contains ten chunks of media page, which will subsequently make up ten RTP packets. Buffer B is filled

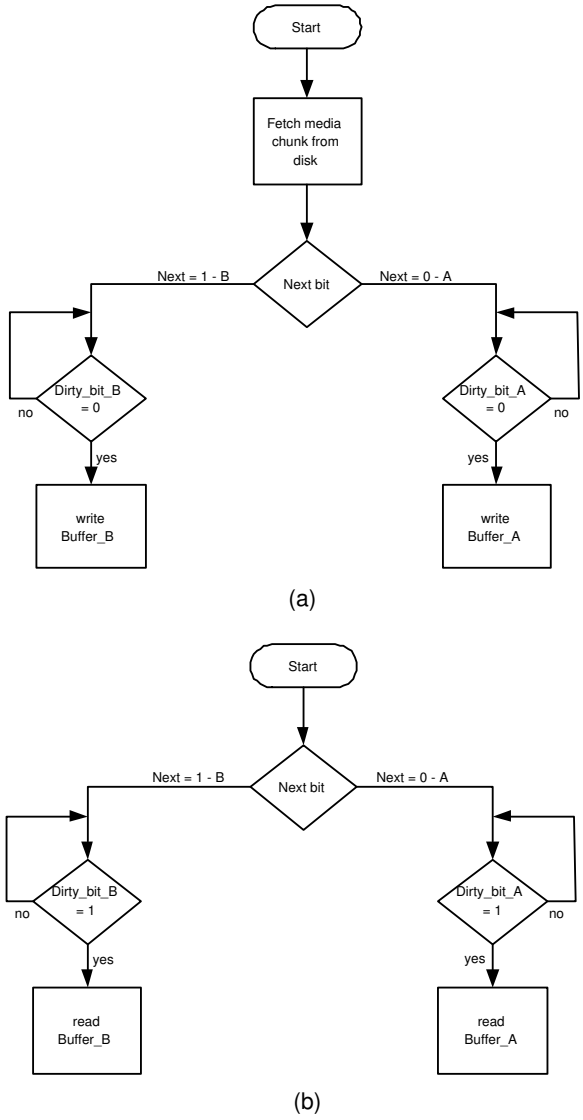


Figure 5.2: Illustration of double buffering (a) Writing to the double buffer (b) Reading from the double buffer.

in a similar fashion. The freshness of a buffer is indicated by a dirty bit. When the dirty bit for a buffer is set, it means the buffer is ready for reading. However, when the bit is reset, the RTP packetizer cannot read the buffer. While the RTP packetizer accesses the buffer according to the rate at which the packets are streamed to the client, which depends on the encoding rate, the buffer is filled as fast as the disk access latency allows. Therefore, the RTP packetizer cannot wait for buffer access, thereby eliminating the possibility of jitter being introduced due to disk access latency variation when the number of clients is large and disk bandwidth becomes a limiting factor.

Filling of the buffer is done as follows: a bit, called the next bit, determines the order in which the filling of the buffer proceeds. When buffer A is filled with the first ten pages, next bit is set and subsequent ten pages are buffered in B after which the bit is cleared. Buffer A is next refilled and the process continues until the entire media object is streamed to the client or the media transaction is terminated.

5.3 Implementation

We implemented the DB-RTP server for Linux platforms. Porting to other unix versions should not require any significant changes in the code, especially if POSIX threads are supported. The server responds to multiple clients by creating threads to handle different connections. To hide disk access latency, we employ a separate

thread to handle disk access for each stream. In other words, each media stream is served by two threads. One thread pre-fetches a media chunk from the disk and buffers it into the memory, while the main thread reads a chunk from the memory and streams it to the client. This prevents the main thread from blocking due to disk latency.

Apart from standard C libraries and system calls, we did not utilize any commercial or free RTP/RTSP libraries e.g., [68]. Working with the raw C libraries and system calls enables us to address performance issues that we can even fine tune.

As explained previously, our prototype is multithreaded. We use the POSIX thread library. Our choice of threads over a multi-process based server is due to the low overhead of thread creation compared to processes. In addition, shared address space among threads simplifies the design and implementation.

In Figure 5.3, we show a screen shot of our RTP packets using a protocol analyzer, *Ethereal* [69]. We use *Ethereal* to capture some packets while the server is streaming media to clients. We show the IP packets and the UDP segment containing our RTP packets.

5.4 Experimental Setup and Load Tool

The experimental setup and load tool used in memory performance evaluation of the prototype DB-RTP server is presented in this section.

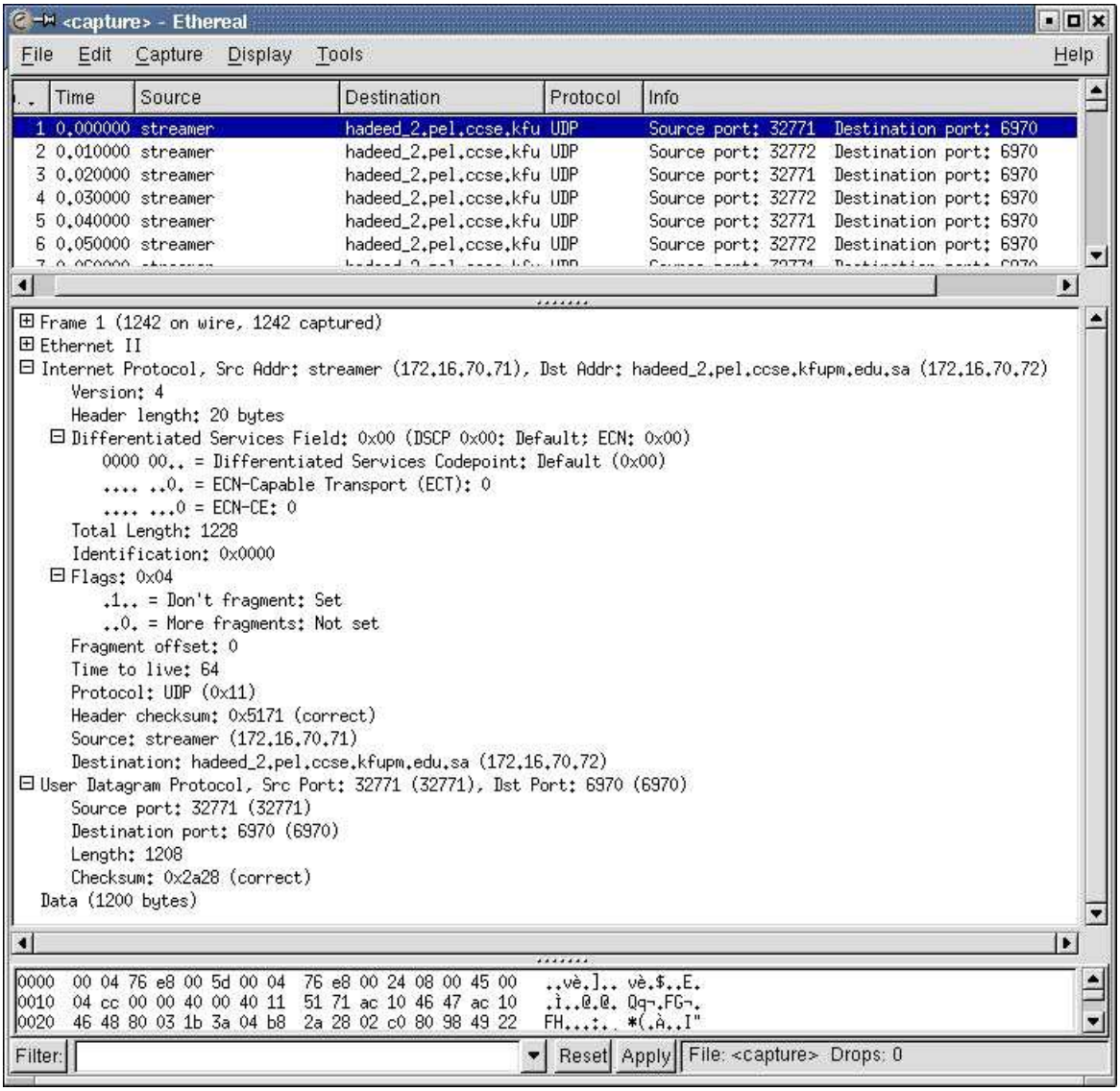


Figure 5.3: Screen shot of DB-RTP server packets captured during streaming session.

5.4.1 Experimental Setup

The experimental setup for the performance evaluation of our prototype DB-RTP server is identical to the experimental setup used for streaming media servers described in Chapter 4. We use same performance metrics. However, we maintain a unique encoding rate of 300 Kbps as this encoding rate has more impact on performance compared to 56 Kbps.

For the purpose of comparison, we designed another RTP server with same functionality as the DB-RTP server, except that the RTP server does not incorporate any enhancement on pre-fetching and buffering. While each stream in DB-RTP server utilizes two threads to hide disk access latency, the RTP server uses a single thread to serve clients. We implement this RTP server to evaluate the performance impact of pre-fetching and buffering

5.4.2 Load Tool

We design a streaming load tool that is capable of generating a large number of clients to request streaming media objects from the server. This tool generates large volume of clients requesting streaming media objects by spawning multiple threads. The number of clients generated and the stream distribution – either all clients requesting same media object or multiple objects – can be specified at command line. Our load tool economizes the use of client machine resources; hence

it is capable of generating a large number of client requests simultaneously. We also incorporate client-side logging of streaming statistics in terms of number of packets lost and jitter.

5.5 Performance Evaluation

In this section we discuss the results of measurement-based performance evaluation of the prototype DB-RTP server. We use RTP server implementation without enhancements – prefetching and buffering – to serve as a base line for performance. We use four alternative implementations for this comparison: (1) RTP-unique, (2) RTP-multiple, (3) DB-RTP-unique, and (4) DB-RTP-multiple. Here unique and multiple refer to the streaming object that the server serves to its clients.

5.5.1 Cache Performance

Figure 5.4(a) compares L1 misses for four server implementations: RTP-unique, RTP-multiple, DB-RTP-unique, and DB-RTP-multiple. Similar to our previous results, the cache misses generally increases with the number of clients. L1 cache in Pentium IV is a split cache – data cache only, hence as more clients are requesting media, the amount of data that has to be moved between cache and memory is large. Since the media data has poor temporal locality and little data reuse, we experience large number of cache misses. Also, since DB-RTP prefetching and buffering is

implemented in the memory, we do not expect any improvement in cache behavior over regular RTP server. The L2 cache behavior is shown in Figure 5.4(b). We observe wide variation in L2 cache misses. Since L2 is a mixed cache, this variation can be attributed to movement of media data and program instruction through the cache.

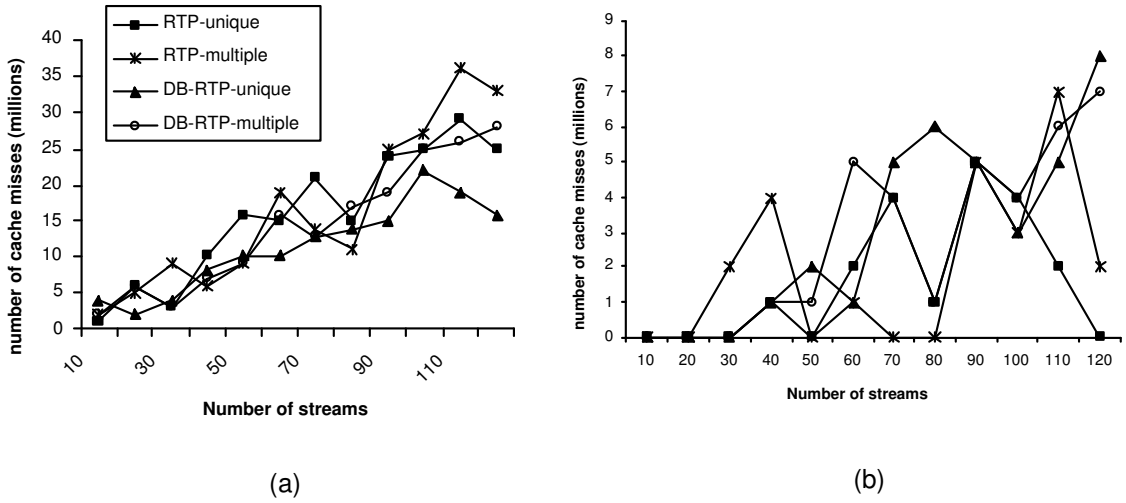


Figure 5.4: Variation of server cache misses with number of clients (a) L1 cache misses (b) L2 cache misses.

5.5.2 Throughput and CPU Utilization

The DB-RTP server shows improvement in throughput delivered to the clients, especially when the number of clients is relatively large. This is shown in Figure 5.5. At large number of clients with multiple stream distribution, disk access latency is likely to be high, hence becoming a potential bottleneck on performance. Prefetching

and double buffering here shows its merit as the bandwidth in this condition is higher than that for the RTP server that does not incorporate prefetching and buffering. We expect even a better performance when multiple disks are used to distribute disks loads.

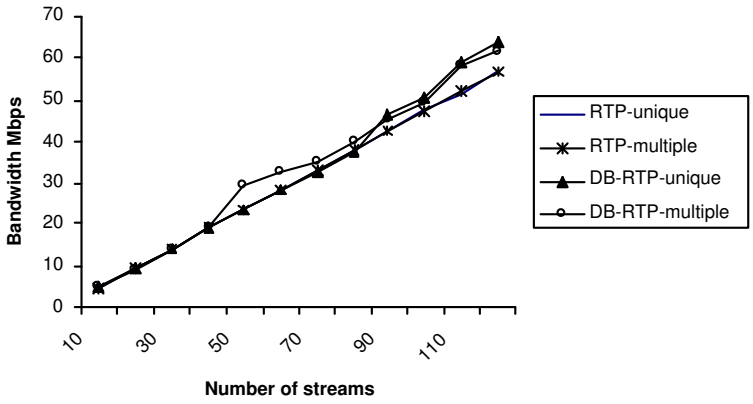


Figure 5.5: Server aggregate throughput (in terms of total bytes transferred per second).

As shown in Figure 5.6, DB-RTP server shows higher CPU utilization. It is intuitive when we have the DB-RTP server using more CPU since it utilizes twice the number of threads used by the regular RTP server. Higher CPU utilization is observed due to greater overhead in context switching of threads. The larger the number of threads we have, the greater the overhead due to context switching.

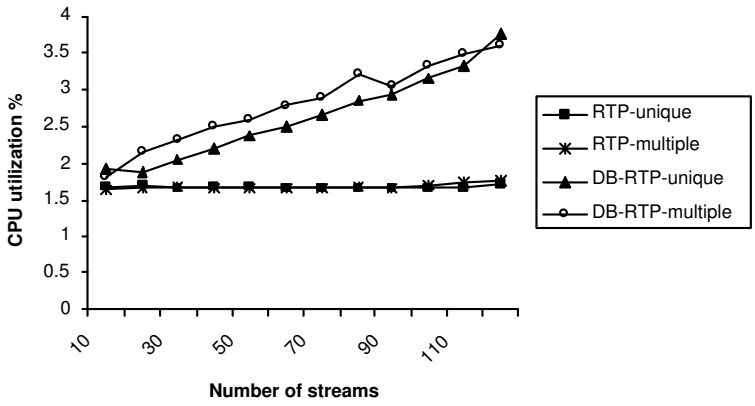


Figure 5.6: Server CPU utilization.

5.5.3 Packet Loss and Jitter

For both servers, we did not record any packet loss. It is not surprising since we are using a closed-LAN operating at 1 Gbps. Probability of packet loss is higher when the packets have to cross a router. In our case, all packets are switched within a single collision and broadcast domain.

We also log inter-packet arrival delay at the clients' end. Figure 5.7 shows the average jitter for the clients. What we refer to as average jitter in this case, is the average of the difference in inter-arrival times of consecutive packets. Media applications expect a uniform and relatively low inter-arrival delay. As shown in the figure, both servers show a uniform average jitter. However, the DB-RTP server show a lower inter-arrival delay. Packet delays can start from the server if the disk access becomes very high to the extent that disk bandwidth cannot sustain the rate at which media object is requested to be streamed. Our pre-fetching and double

buffering alleviates this situation where disk becomes a bottleneck.

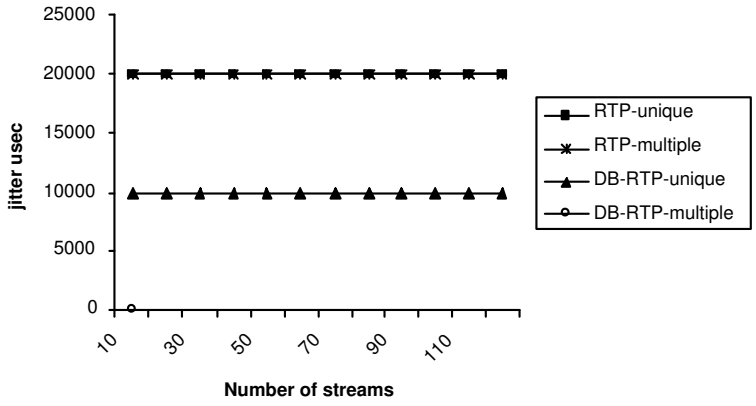


Figure 5.7: Streaming client jitter.

5.6 Summary

L1 cache miss increases with number of clients. On the other hand, L2 cache misses vary in a non-regular manner. While L1 is a split cache for data only, L2 is a mixed cache. The same observation applies to RTP server for cache behavior. Throughput increases with number of clients. While we observe maximum throughput of 63.85 Mbps for DB-RTP server, the observed throughput for RTP server is 59 Mbps. For jitter, both servers exhibit steady jitter, but DB-RTP has relatively lower jitter compared to RTP server.

DB-RTP server shows, though, marginal gain in throughput and jitter, prefetching and buffering would show more advantage when the number of clients is very

large – in the range of thousand clients. However, as the number of clients grow larger, another factor that will negatively affect performance is threads scheduling overheads. We are constrained to use low number of clients in our experiments due to the limitation of the number of threads that can be spawned per process. This is a Linux kernel 2.4 limitation that we hope to address in our future work.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

In this thesis, we thoroughly analyzed the impact of memory subsystem on performance of representative high throughput network devices. We analytically calculated peak throughput with respect to available memory and I/O bandwidth and conducted measurement-based performance studies of key high throughput servers: Apple Darwin streaming server and Windows media server; Apache web server and Microsoft Internet Information server; and Linux software router. We also designed and implemented an enhanced RTP server that uses double buffering to hide latencies due to disk accesses and memory buffering.

The following can be concluded based on the analytical and measurement-based memory performance evaluation reported in this thesis:

- Analytically derived optimistic peak throughputs bounds show that network applications deployed on general-purpose microprocessor based platforms can be enhanced to deliver high throughput if software overhead and related constraints like bus contention are resolved.

- Measurement results show that server throughput is significantly degraded by excessive cache misses and page faults. Penalty of cache misses and page faults are significant because of the resulting high latency in servicing a cache miss or page fault.

- Our prototype shows that performance improvement in throughput and jitter can be gained by implementing efficient memory buffering in streaming media servers.

- We also used microbenchmarks to investigate the contribution of operating system impact on specific aspects of performance. On memory transfer, we find Linux operating system (kernel 2.4.7-10) comparable to Windows 2000, while for context switching overhead, Linux shows higher overhead per context switch.

Memory performance, as we demonstrated in this work, is a key inhibiting factor for throughput of network servers. Any means that will improve memory subsystem performance (on-chip and off-chip caches, main memory, and virtual memory) will bring significant benefit to the performance of these servers. As memory access speed

remains a technology dependent issue, viable solutions such as, memory latency hiding have shown remarkable improvement.

6.2 Open Questions

While this thesis reported analytical as well as measurement based performance evaluation of high throughput servers for network infrastructure applications, it raises several other interesting questions. The rest of this section presents these open questions. We consider them as open questions only because further research efforts are needed to provide concrete answers to these questions.

With specific-purpose architectures like network processor, the system CPU will be relieved from some computational demand [10]. This can be investigated against the use of general-purpose architectures. Network processor for example can reduce contention for the system's internal bus and I/O bus. Use of intelligent network adapters at MAC layer can also be investigated to see their effect on performance of high throughput servers.

How will memory compression affect high throughput servers? Memory compression has been used as a means to enhance system performance [30] Memory compression and decompression is CPU intensive, and how it will affect high throughput servers requires further investigation. For instance, the content served by a streaming media servers is compressed and how significant is the memory compression to

a server that serves already compressed content remains to be seen.

Scheduling issues are important for high throughput servers and require further investigation. Bus scheduling aimed at reducing conflict/contention may impact the performance of high throughput servers. Extent of this impact remains to be evaluated. Determining an optimal number of NICs that can be used for a particular type of bus is another important open question.

6.3 Future Research

A number of related research efforts can benefit from the findings reported in this thesis. We outline some of them in the following:

Server Development

An efficient server design can incorporate polling with multiplexing, and multithreading to minimize process scheduling overhead to the operating system. A server design that uses multiplexing to serve multiple request has low context switching overhead. In addition, multithreading is a useful mechanism for tolerating memory access latency. Using both schemes will yield a hybrid server design that will incorporate the benefits of both the methodologies.

Special Architectures

Special architectures like network processors and other ASIC-based devices can be investigated for specific server applications to boost performance. The areas that can be investigated further include:

- Resource scheduling – new mechanisms and algorithms for CPU and bus contention based scheduling optimized with respect to minimum contention for bus rather than efficient CPU use.
- Investigation of the role of I/O overhead, in addition to the memory overhead.
- Use of IRAM (intelligent RAM) architectures [70, 71].
- Implementation of an integrated network infrastructure server that includes low-level IP packet forwarding as well as higher-layer level value-added service processing like differentiated service (DiffServ), webmulticast, proxy, network address translation (NAT), voice over IP (VoIP) gateway, etc.

In contrast to the above open questions and future research, this thesis has thoroughly analyzed the fundamental high throughput networking server issue: the memory performance. Many other server performance issues, such as I/O performance, use of IRAM architectures, or bus scheduling will benefit from the understanding of memory performance gleaned from this work.

Bibliography

[1] M. Reisslein, F. Hartanto, and K. Ross. Interactive video streaming with proxy servers. *INFOCOM*, 2000.

[2] S. Sahu, P. Shenoy, and D. Towsley. Design considerations for integrated proxy servers. *Proc. of International Workshop on Network and Operating Systems Support for Digital Audio and Video*, June 1999.

[3] J. L. Hennessy and D. A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.

[4] Prashant Pradhan and Tzicker Chiueh. Implementation and Evaluation of QoS-Capable Cluster-Based IP Routers. *In Proceedings of High Performance Networking and Computing Conference (SC'02), Baltimore, MD*, November 2002.

[5] Francky Catthoor and Nikil Dutt. Hot topic session: How to solve the current memory access and data transfer bottlenecks: at the processor architecture or

at the compiler level? *Proc of Design Automation and Test in Europe*, March 2000.

[6] C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 31(7):23–32, January 1998.

[7] C. Kozyrakis and D. Patterson. A new direction in computer architecture research. *IEEE Computer*, 31(11):24–32, November 1998.

[8] J. Carter and et al. Impulse: Building a smarter memory controller. *In 5th Int. Conference on High Performance Computer Architecture*, pages 70–79, January 1999.

[9] Prashant Pradhan and Tzicker Chiueh. Implementation and Evaluation of A QoS-Capable Cluster-Based IP. *In Proceedings of IEEE SuperComputing*, November 2002.

[10] Douglas E. Comer. *Network Systems Design: Using Network Processors*. Pearson Prentice Hall, New Jersey, 2004.

[11] Dapeng Wu, Yiwei Thomas Hou, Wenwu Zhu, Ya-Qin Zhang, and Jon M. Peha. Streaming video over the Internet: Approaches and directions. *IEEE, Transactions on Circuit and Systems for Video Technology*, 11(3):282–300, March 2001.

[12] Requirements for IP Version 4 routers. *Internet Engineering Task Force, RFC 1890*, June 1995.

[13] Terry Dawson. Building high performance linux routers. http://linux.oreillynet.com/pub/a/linux/2000/09/28/LinuxAdmin.html, 2000 September.

[14] C. Thomborson and Y. Yu. Measuring data cache and TLB parameters under Linux. Technical Report, 2000.

[15] M. E. Wolf and Monica S. Lam. A data locality optimization algorithm. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 26(6):30–44, June 1991.

[16] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. *Proceedings of International Conference on Supercomputing*, pages 238–253, July 1998.

[17] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[18] M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance of blocked algorithms. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[19] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. *CWI Technical Report INS-R0203*, August 2002.

[20] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Eng*, 14(4):709–730, July 2002.

[21] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246,, December 2000.

[22] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a Join? - Dissecting CPU and memory optimization effects. *In Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 339–350, September 2000.

[23] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. *In Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65, September 1999.

[24] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. *Proceeding of the Workshop on Interaction be-*

tween Compilers and Computer Architectures, Third International Symposium High-Performance Computer Architecture (HP CA-3), February 1997.

[25] F. Douglis. The compression cache: Using on-line compression to extend physical memory,. *In Winter 1993 USENIX Conference USENIX Assoc*, pages 519–529, 1993.

[26] M. Kjels, M. Gooch, and S. Jones. Modeling the performance impact of main memory data compression. *Proceedings 12th UK Computer and Telecommunications Performance Engineering Workshop*, pages 169–184, September 1996.

[27] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. *In Proceedings of 1999 USENIX Annual ConferenceUSENIX Assoc*, pages pages 101– 116, June 1999.

[28] J.S. Lee, W. K. Hong, and S. D. Kim. Design and evaluation of a selective compressed memory system. *In IEEE International Conference on Computer Design*, pages 184–191, 1999.

[29] C. Benveniste, P. Franaszek, and J. Robinson. Cache-memory interfaces in compressed memory systems. *Technical Report RC 21662, IBM Research Division, T.J. Watson Research Center*, February 2000.

[30] S. Roy, R. Kumar, and M. Prvulovic. Improving system performance with compressed memory. Technical Report, 2001.

[31] J. B. Carter, W. C. Hsieh, L. B. Stoller, M. Swansony, L. Zhang, and S. A. McKee. Impulse: Memory system support for scientific applications. *Scientific Programming, IOS Press*, 7(3-4):195–209, fall 1999.

[32] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and Sally A. McKee. The impulse memory controller. Technical Report, 2001.

[33] P. J. Shenoy, P. Goyal, and H. Vin. Issues in multimedia server design. *ACM Computing Surveys*, 27(4), December 1995.

[34] J. M. Sohn, G. Y. Kim, and T. G. Kim. Performance measurements of a small-scale VOD server based on the UNIX. *The Third IEEE Symposium on Computers and Communications ISCC'98 Athens, Greece*, June 1998.

[35] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz. A disk-based storage architecture for movie on demand servers. *Information Systems*, 20(6):465, 1995.

[36] B. Sonah, M.R. Ito, and G. Neufeld. The design and performance of a multimedia server for high-speed networks. *Proceedings of IEEE International Conference on Multimedia Computing and Systems, ICMCS*, 1995.

[37] M. Weeks, H. Batatia, and R. Sotudeh. Improved multimedia server I/O subsystems. *Euromicro98, 24th Conference Proceedings*, 1998.

[38] M. Weeks and C. Bailey. Continuous discrete-event simulation of a continuous-media server I/O subsystems. *Euromicro 2000, Workshop on Multimedia and Telecommunications*, September 2000.

[39] A. L. Reddy and J. Wyllie. IO issues in a multimedia system. *IEEE Computer*, 27(3):69–74, March 1994.

[40] S. Rixner. *A Bandwidth-efficient Architecture for a Streaming Media Processor*. Phd thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 2001.

[41] P. Shenoy and H. M. Vin. Efficient striping techniques for multimedia file servers. *Performance Evalution*, 38(3-4):175–199, December 1999.

[42] S. Berson, L. Golubchik, and R. R. Muntz. Fault tolerant design of multimedia servers. *In Proc. ACM SIGMOD'95*, pages 364–375, May 1995.

[43] R. Tewari, D. M. Dias, W. Kish, and H. Vin. Design and performance tradeoffs in clustered video servers. *In Proc. IEEE International Conference on Multimedia Computing and Systems*, pages 144–150, June 1996.

[44] B. Ozden, R. Rastogi, P. Shenoy, and A. Silberschatz. Fault-tolerant architectures for continuous media servers. *In Proc. ACM SIGMOD'96*, pages 79–90, June 1996.

[45] A. Mourad. Doubly-striped disk mirroring: Reliable storage for video servers. *Multimedia, Tools and Applications*, 2:253–272, May 1996.

[46] J. Gafsi and E. W. Biersack. Performance and reliability study for distributed video servers: Mirroring or parity? *In Proc. IEEE International Conference on Multimedia Computing and Systems*, pages 628–634, June 1999.

[47] Asit Dan, Daniel M. Dias, Rajat Mukherjee, Dinkar Sitaram, and Renu Tewari. Buffering and caching in large-scale video servers. *In Proceedings Compcon*, pages 217–224, March 1995.

[48] Banu Ozden, Rajeev Rastogi, Avi Silberschatz, and Cliff Martin. Demand paging for video-on-demand servers. *International Conference on Multimedia Computing and Systems (ICMCS'95)*, May 1995.

[49] Asit Dan. Buffer management policy for an on-demand video server. Technical Report, 1997.

[50] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. *In Proceedings of the ACM Multimedia*, pages 391–398, 1994.

[51] Dario Luparello, Sarit Mukherjee, and Sanjoy Paul. Streaming media traffic: an empirical study. *6th International Workshop on Web Caching and Content Distribution*, June 2001.

[52] M. F. Arrit and C. L. William. Web server workload characterization: The search for invariants. *ACM SIGMETRICS'96*, 1996.

[53] K. Krishna and M. Prasant. Architectural impact of secure socket layer on internet servers. *International Conference on Computer Design (ICCD)*, September 2000.

[54] A. Goldberg, R. Buff, and A. Schmitt. Secure web server performance dramatically improved by caching SSL sessions keys. *Workshop on Internet Server Performance, SIGMETRICS'98*, June 1998.

[55] Arun Iyengar, Ed MacNair, and Thao Nguyen. An analysis of web server performance. *In Proceedings of the IEEE 1997 Global Telecommunications Conference (GLOBECOM'97)*, November 1997.

[56] James C. Hu, I. Pyarali, and Douglas C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. *In Proceedings of GLOBECOM'97*, 1997.

[57] Amisu O. Salam-Alada and Abdul Waheed. Performance comparison of Apache and Microsoft IIS web server. *In Proceedings of International Arab Conference on Information Technology*, December 2002.

[58] Vittorio Trecordi and Alberto Verga. An experimental study on the performance of www servers. *Global Telecommunications Conference, GLOBECOM'96*, pages 22–27, November 1996.

[59] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. *17th ACM Symposium on Operating Systems Principles (SOS '99) - Operating Systems Review*, 34(5):217–231, December 1999.

[60] Eddie Kohler. *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, February 2001.

[61] Oscar-Ivan Lepe-Aldama and Jorge Garcia-Vidal. A performance model of a PC based IP software router. *IEEE ICC*, 2002.

[62] Xiaohu Qie, Andy Bavier, Larry Peterson, Scott, and Karlin. Scheduling computations on a software-based router. *In Proc. IEEE Joint International Conference on Measurement Modeling of Computer Systems (SIGMETRICS)*, June 2001.

[63] Jason R. Hess, David C. Lee, Scott J. Harper, Mark T. Jones, and Peter M. Athanas. Implementation and evaluation of a prototype reconfigurable router. *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1999.

[64] Ch. Kurmann and T. Stricker. Characterizing memory system performance for local and remote accesses in high end SMPs, low end SMPs and clusters of SMPs, year=1998. *7th Workshop on Scalable Memory Multiprocessors held in conjuction with the 25th Annual International Symposium on Computer Architecture ISCA'98*, June.

[65] Bradford G. Edward. High-performance programming techniques on linux and windows. http://www-106.ibm.com/developerworks/linux, July 2002.

[66] G. Trent and M. Sake. Webstone: The first generation in http server benchmarking. Available: http://www.sgi.com/Products/Web-FORCE/WebStone.

[67] Hewlett-Packard Company, Information Networks Division. *Netperf: A Network Performance Benchmark*, February 1996.

[68] Florin Lohan, Irek Defee, and Marius Vlad. The architecture of an integrated RTSP, RTP and SDP library.

[69] Gerald Combs. *Ethereal network protocol analyzer.* http://www.ethereal.com.

[70] Brian R. Gaeke, Parry Husbands, Xiaoye S. Li, Leonid Oliker, Katherine A. Yelick, and Rupak Biswas. Memory-intensive benchmarks: IRAM vs. Cache-based machines. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS). Ft. Lauderdale, FL*, April 2002.

[71] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. Vector IRAM: A media-oriented vector processor with embedded DRAM. *12th Hot Chips Conference, Palo Alto, CA*, August 2000.

Vitae

- **Garba, Ya'u Isa**
- Born in Nigeria.
- Received Bachelor of Engineering degree in Electrical and Electronic Engineering from Abubakar Tafawa Balewa University Bauchi, Nigeria, in September 1998.
- Joined Computer Engineering Department, KFUPM, as a Research Assistant in January 2001.
- Received Master of Science Degree in Computer Engineering from KFUPM, Dhahran, Saudi Arabia in June 2003.