The Design and Performance of a Bare PC Web Server

Long He, Ramesh K. Karne, and Alexander L. Wijesinha

Department of Computer and Information Sciences, Towson University, Towson, MD 21252

{lhe, rkarne, awijesinha}@towson.edu

Abstract

There is an increasing need for new Web server architectures that are application-centric, simple, small, and pervasive in nature. In this paper, we present a novel architecture for a bare PC Web server that meets most of these requirements. A bare PC Web server runs on any Intel 386 (or above) based architecture with no operating system in the machine. The application object or program contains all the necessary code to run in the bare machine including its boot and load programs. This approach to building Web servers has many notable features including optimized task scheduling, zero copy memory buffering, minimal interrupt intervention, concurrency control avoidance, minimal resource dependence, and potential scalability via a cluster of bare PC Web servers. The bare PC Web server executable is small, self-contained, and fits on a single floppy disk. The server does not use a local hard disk and the resource files are located on the network. We compare performance measurements for the bare PC Web server with three other optimized commercial Web servers: Microsoft IIS, Apache, and Tux. It is found that the bare PC Web server achieves an average first response time that is 3.9 – 5.4 times better, and a connection time that is 1.4 – 2.8 times better than the other servers. We also discuss alternative designs for bare PC Web servers and compare their performance.

Keywords: DOSC, Web Server, Bare PC, AO, Performance.

1. Introduction

With the increasing reliance on the Internet worldwide, there is a need for innovative Web server architectures that continue to perform well under higher traffic demands, allow easy maintenance, do not depend upon fast changing environments, and run on a variety of devices. Popular Web servers support millions of users who access the Internet daily for a variety of reasons including reading the news, searching for information, or making a purchase. To cope with the demand, Web sites such as Google [4] employ a cluster of Web servers to handle large volumes of traffic. Many users also choose to have their own Web sites hosted on a server run by their Internet Service Provider (ISP).

In addition to running on high-end server machines and ordinary desktops, Web servers run on network devices such as switches or routers, and on network appliances, enabling technicians or users to remotely monitor these devices using a browser. Given these diverse needs, it would be useful to create a generic and cheap Web server architecture and design for multiple application domains.

Extensive research on Web servers has been undertaken in the past, and we will only give a few examples here. The impact on Web server performance of new socket functions, per-byte optimizations (to reduce or eliminate data reading operations), and per-connection optimizations (to reduce the number of packets exchanged during connection establishment or termination) is investigated in [27]. Buffering and caching systems by themselves are shown to improve Web server performance by 40 to 80% [24, 28]. Other directions in Web server research include the use of a simple embedded Web server library (SWILL) to build Web applications [25], the AutoTune agent framework based on intelligent agents and control theory for automatic performance tuning of a Web server [7], and queuing models to study the behavior of the Apache Web server as in [1].

There are also many projects that focus on the operating system (OS) with a view towards improving application performance or giving more control to applications. Server operating systems using exokernel [17] demonstrate the feasibility of building servers at the application level. For example, by moving networking functions to applications, it was shown that application-level servers can achieve 3-8 times more throughput than servers running on OpenBSD systems [13]. However, it should be noted that in these studies the exokernel was compared with much less optimized OpenBSD systems resulting in a dramatic improvement in performance. Today's Web servers are designed for performance and implement many optimization techniques such as single copy buffering, direct API to networking, direct access to Ethernet buffers, caching Web pages, and other performance enhancements, and so it is not possible to achieve performance levels similar to exokernel in comparative studies of Web servers. The Fluke kernel [11] provides an API characterized by fully interruptible and restartable ("atomic") kernel operations that has many advantages for applications. Application-specific operating systems and embedded systems [12] generate an OS for a given application, thus avoiding unnecessary operating system functions. IO-Lite [28], Flash [29] and bare-metal Linux [35] attempt to directly communicate with the hardware by using special mechanisms in the kernel to bypass the OS. The Tux Web server, which is integrated into the Linux kernel and incorporates several optimizations, improves its throughput by up to 36% as a result of fine-tuning its strategy for accepting connections under conditions of high load [6].

Most research prototypes and commercial Web server architectures, including the above, are based on some form of an underlying operating system. The significant improvement in Web server performance using exokernel resulted from moving network operations into the application [13]. However, this approach still requires some OS components. Similarly, the OSKit approach [31] avoids using a conventional OS but needs kernel libraries. Most embedded and mobile servers also require an OS, such as Tiny OS [32], which is embedded with its application. The removal of all operating system abstractions and operating system inefficiencies is discussed in [9, 10].

In contrast to previous Web server architectures that retain at least some elements of an OS, we propose a novel approach to building Web servers that completely eliminates the OS. The basic idea is to disperse the necessary operating environment functionality into applications, which results in the dispersed operating system computing (DOSC) paradigm [18]. DOSC is a general purpose computing paradigm that decouples hardware and software through *direct application programming interfaces (API) to the hardware*. As such, it is not an embedded approach. In DOSC, there is no software running in a system other than the application itself, which is in total control of the hardware.

It might seem that this approach is similar to the one used in the early days of computing when application programs could not be decoupled from the underlying system and hardware. However, the DOSC paradigm differs from the historical approach to computing as it is based on the premise that today's computer memory sizes could be close to the logical address space (for example, 4 gigabytes in 32-bit systems) and software is capable of communicating to the hardware through direct APIs. It will be seen in Section 3 that the direct API to hardware in DOSC is not too complex and reasonably easy to use. The present technological advancements make it possible to use the DOSC paradigm to build computer applications that are lean, efficient, and application-centric. This approach could not be have been used with the large non-programmable hardware units and primitive technology of the past.

Today’s application specific integrated circuits (ASIC) and embedded systems are different from the DOSC paradigm. An ASIC does not provide any general purpose API for application programs to execute in its environment. Instead, it is tightly integrated with a given application and hardware. An embedded system is similar in that it includes a lean operating system kernel with its application. The cell phone and iPod are examples of embedded and ASIC systems, which clearly have no API. The DOSC paradigm is an application programming concept wherein applications can communicate with the hardware directly. The hardware API in DOSC could be integrated with the hardware in the future. At present, we encapsulate the API into its application object, which allows the bare hardware to be used without any changes.

When the necessary operating environment is dispersed into an application program, the application programmer has to be more knowledgeable about the hardware. In effect, it requires the programmer to be both a systems programmer and an application programmer. Thus, in the DOSC paradigm, we are trading off less system complexity and improved performance versus increased programmer skills and knowledge.

In this paper, we describe a bare PC Web server that is designed and implemented using the DOSC paradigm, and compare its performance with three optimized commercial Web servers that run on a conventional OS. As Web servers are domain-specific applications and designed for high performance, there is no need to provide any non-essential operating system features including script files, non-administrative logins, and debug facilities. Thus, a Web server application can be designed as a domain-specific Application Object (AO) [20], which consists of all the code necessary to run the application. This includes code for booting, loading, task switching, I/O management, and networking. The AO concept is analogous to service-oriented computing [30], in which service-oriented application-specific objects perform relevant Web services; however, unlike an AO, service-oriented objects do not carry the entire computing environment needed to run on a bare machine. Service-oriented computing is also a paradigm that utilizes services as the constructs to support the development of rapid, low-cost and easy composition of distributed services. In contrast, an AO encapsulates its services within itself and communicates with other AOs through message passing. The bare PC Web server application is a single AO that runs on a bare PC without any OS or other software. The general AO architecture concept was introduced in [21, 22].

The architecture of the bare PC Web server is simple and robust, and does not require any other environment such as an OS, or boot and load facilities. It can be made pervasive, if the underlying device uses an Intel 386 (or above) based processor architecture (or some standard CPU architecture) for the currently designed API. To maintain simplicity, the Web server AO uses the keyboard, display, floppy drive, and an Ethernet connection. It does not use any other I/O. As a bare PC system does not use a hard disk, only allows connections to port 80, and does not allow software other than its AO to run on the system, it is attractive from a security viewpoint. Moreover, it is capable of high performance in spite of its simplicity. Bare PC applications are also small in size, extensible, flexible, and easy to maintain. A final point to note is that many of the general techniques proposed in the literature for improving Web server performance, some of which were mentioned above, can be easily implemented in the bare PC Web server to further improve its performance.

The rest of this paper is organized as follows. Section 2 describes the architectural components of a bare PC Web server. Section 3 provides more details on its design and implementation. Section 4 contains performance measurements comparing a bare PC Web server with IIS, Apache [2] and Tux [34] Web servers, and also comparing alternative designs of bare PC Web servers. Section 5 contains the conclusion.

2. Architecture

The novel bare PC Web server architecture differs in many respects from that of a conventional Web server running on an OS. The server design is based on the DOSC paradigm implying that only the Web server AO runs on the system, while no other application runs at the same time (unless the applications are closely related to each other such as a Web browser, an email client, and a VoIP soft-phone running as a single AO on a bare PC). The basic hardware elements currently used in the bare PC Web server are CPU, memory, floppy drive, Ethernet network interface card (NIC) or onboard network chip (3Com 905CX or Intel 82540EM), keyboard, and display. The Web server AO controls the CPU and memory available in the bare PC (there is no OS). Thus, an application programmer designs and builds this AO, and decides on the CPU and memory requirements needed for the application.

In fact, the owner of the Web server has total control of the application on a bare PC and manages the above mentioned resources for which we have developed standard interfaces and drivers. We have written C++ interfaces (an API) [19] for managing tasks, interrupts, exceptions, and debugging facilities. As these interfaces are simple and robust, we are able to run a variety of common applications on a bare PC including an editor, email and VoIP softphone [23]. (A bare PC running the latter application connects to the Internet in the usual manner through an Internet

Service Provider at home, or through an office or campus LAN, providing peer-to-peer communication without requiring a third-party server or any other software.)

A bare PC and a conventional PC connected to the router can coexist serving their own applications without interference. All users can thus run their own bare PC Web servers (and host their Web sites) at any location without the need for a platform or an environment. A bare PC Web server can also be used for other custom applications under the control of the user as the Web server AO and all the resources of the bare PC are owned by the user. The entire code for the Web server AO (including an implementation of the necessary communication protocols) fits on a single floppy disk (bare PC applications can also be made to work with flash memory).

The main elements of the bare PC Web server architecture are shown in Figure 1. HTTP requests that have arrived are placed in the Ethernet buffer and the device driver for the Ethernet card (or chip) stores them in a circular list known as an upload pointer descriptor (UPD). Likewise, responses (data) leaving the Web server are kept in a circular list known as a download pointer descriptor (DPD) by the application. The UPD and DPD structures are located in user memory. In our system, user space and system space are one and the same as the AO programmer controls the memory map for the entire application.

Figure 2 illustrates the task structure for the Web server, which consists of three types of tasks: Main, RCV and HTTP. The RCV task and the HTTP task return back to the Main task whenever they are idle. After the system starts, it runs the Main task. A receive task (RCV task) waits for a message to be placed in the UPD (receive list). When a message arrives, the RCV task will handle the request through the phases of Ethernet, IP/ARP, and TCP processing (an ARP request is handled separately and is not shown in the figure). It processes a TCP message by creating a new table entry in the TCB (TCP control block), or updating an existing entry in the TCB. The TCP processing is done to either establish a connection (SYN, SYN-ACK, and ACK messages), or to process a GET message by initiating an HTTP request (if the connection is already established). The initiation of an HTTP request involves popping an idle task from the stack consisting of a task pool and inserting it into the run list as an HTTP task. The HTTP task does the TCP, IP, and Ethernet processing needed to send the requested file to the client.

A separate HTTP task is invoked for each client request and its TCB entry is updated once a GET message is received by the server. Thus, there can be many concurrent HTTP tasks running in the system. As TCP requires a task to be suspended for the duration of an RTT (round-trip-time), the tasks are interleaved in time to provide a maximum level of concurrency. After sending the data, connection termination is done as usual through TCP as shown in Figure 3 (FIN, FIN-ACK, and ACK messages), and the entry in the TCB table is deleted thus completing a single client request. When the Web server is loaded, there will be a Main task running in the system. When a client (HTTP) request arrives, the RCV task will be running, and when it is done as described above, it will return to the Main task. Similarly, when a GET request arrives, the RCV task initiates an HTTP task, so it can be placed in the run list. When choosing between the two types of tasks, the RCV task always has higher priority. There could be many HTTP tasks in the run list (each associated with a client request) waiting to be processed. They are handled on a first-come-first-serve (FCFS) basis. Whenever an HTTP task gets suspended (RTT delay), it will return to the Main task; and when a HTTP task is done processing a client request, it will be placed in the stack pool of idle tasks. The Main task is more like a dispatcher and the time spent in this task can be considered as CPU idle time.

The Web server architecture and task structure is simple as it is customized for this application. There is no unnecessary functionality in the system and no inherent system complexity due to an underlying OS. Some significant novel features of this architecture are described below.

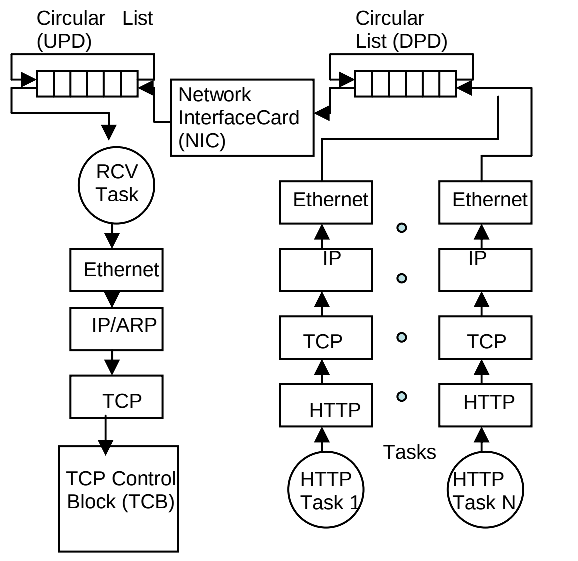

Figure 1. Web Server Architecture

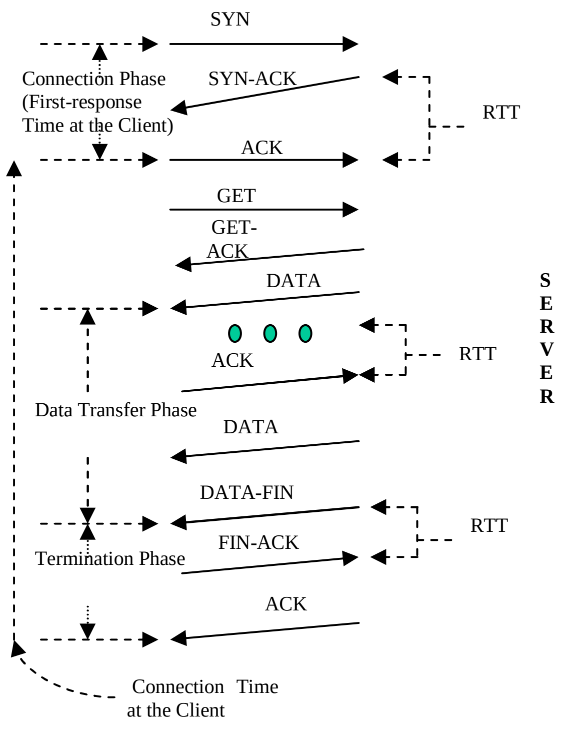

Figure 3. TCP Message Exchange

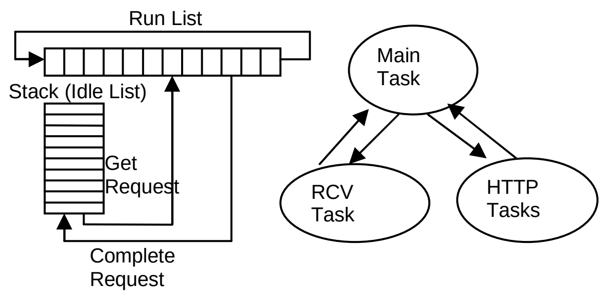

Figure 2. Web Server Task Structure

2.1 Network Protocol Design

Cross-layer design [8], which is becoming popular in wireless networks, enables traditionally independent protocol layers to share information and optimize performance. Protocol design in a bare PC Web server is similar to cross-layer design in that it is not bound by strict network layering conventions. As shown in Figure 1, when a packet arrives in the UPD buffer it can be accessed by the TCP layer for processing in one step. Similarly, when the TCP layer sends a segment, the pointer for the packet is directly placed in the DPD buffer. In general, layers are designed as objects that can be invoked by a single higher level protocol to add lower layer headers to outgoing data or to strip off all headers from incoming data. Header manipulation is efficient since it is done by the RCV task and HTTP tasks in a single step before returning to the Main task.

Furthermore, the bare PC Web server implements lean versions of networking protocols and uses additional techniques to improve protocol performance. For instance, it includes a TCP optimization in which the FIN is piggybacked with the last data packet [27], as shown in Figure 3. In view of the simple bare PC protocol design, it would be easy to add other enhancements such as the loss recovery and congestion control mechanisms included in the TCP-INT implementation [3].

2.2 Optimized task scheduling

Initially, the Web server was equipped with a 3Com 509B external Ethernet card. We used interrupts and interrupt service routines to process HTTP requests; and round robin scheduling for task management [14]. The Ethernet driver for this card does not support the storing of data packets in user memory (no UPD and DPD buffers), and requires synchronous interrupts to send and receive messages. The original prototype was developed to demonstrate the feasibility of a bare PC Web server (it was not a performance driven design). When we attempted to build a "close to optimal" Web server on a bare PC, we found that traditional interrupt driven mechanisms and scheduling techniques suitable for OS-based systems do not yield the best performance for a bare PC. In the latter, task scheduling is controlled by an AO i.e., an AO programmer decides on the task structure during AO construction. In this case, novel approaches to scheduling become possible. When an application is written, an AO programmer has prior knowledge of the application's behavior, and a simple scheduling approach can be used instead of a general-purpose mechanism. The principle that justifies the scheduling approach in a DOSC machine is the following: when a task is not doing useful work (CPU processing only), then it should return to the Main task. In the Web server, the RCV task will return to the Main task after processing a received request. Similarly, an HTTP task will return to the Main task if it is suspended. Thus, the entire scheduling issue becomes simple and results in a task structure that is optimized for a Web server. The performance measurements given in Section 4 reflect the optimality of our task scheduling strategy.

2.3 Zero copy memory buffering

In a conventional system, the user space is virtual and controlled by an operating system. Data stored in an application program's buffer needs to be transferred to an Ethernet card buffer, and vice-versa. Modern Ethernet cards allow their buffers to reside in user address space. In conventional systems, it is sometimes possible to do one copy between buffers [27]. However, such techniques require special instruction support, or shared memory [5] from the OS, which are not available in all systems. Web servers such as IO-Lite [28] or Tux [34] claim to achieve zero copy memory buffering, but they require specific interfaces and modifications to cut through OS layers introducing complexity in design and operation. Relevant issues that need to be addressed in this context include immutable buffers, buffer aggregate support, impact of virtual memory, paging, caching, cache validation and replacement, file cache, and early de-multiplexing in the Ethernet driver [28].

The bare PC Web server architecture inherently provides a zero copy buffering scheme without any special efforts. The Ethernet card stores the incoming data packets in the UPD buffers asynchronously until the circular list buffers are full. The RCV task processes a packet from the UPD buffer and keeps it in the buffer for as long as necessary. The resource files are stored in memory to serve client requests; during the send process, the DPD data buffer pointers are linked with the resource file data pointers in memory and kept until a transmit packet interrupt arrives from the Ethernet card. The DPD and UPD buffers are circular lists with a maximum size of 10,000 (this size was found to be sufficient during maximum stress tests). Thus, copying data is avoided during Web server operation in this architecture.

2.4 Minimal interrupt intervention

As mentioned earlier, we initially investigated a bare PC Web server architecture based on interrupts. OS-based Web servers use interrupts to indicate that a packet has arrived, or to indicate that a packet is available for transmission. They commonly use an interrupt service routine to process a request as soon as it arrives. This approach (as opposed to polling) obviously provides a better response time. However, if the packets arrive at a faster rate, then there is a large interrupt processing overhead, and task processing is delayed. Interrupts are needed in a conventional system, because the system is running many other unrelated tasks, and a given application has no control. In a bare PC, an AO has total control of the machine, and each task is always performing productive work (otherwise, it returns to the Main task). Thus, if the Main task checks for message arrivals and dispatches the RCV task, and also checks for pending

HTTP requests and schedules an HTTP task, it is possible to achieve optimal performance in the Web server. By avoiding receive interrupts for the NIC, and processing transmit interrupts to acknowledge the interrupt controller, we found that Web server performance was much better than with interrupts, and furthermore, the resulting design is much simpler. The only interrupts needed in the bare PC Web server are transmit interrupts from the NIC, and hardware timer interrupts for measuring time.

2.5 Concurrency control avoidance

In the preliminary version of the bare PC Web server [14], many data structures and resources for serving client requests were shared. This approach required locking and synchronization points, and resulted in the serialization of tasks. In addition, locking contributed additional complexity when coding and testing the server. In a bare PC Web server, client requests are independent of each other, and each request goes through successive protocol processing; once a request is done, the entry is removed from the TCB table. Thus, it is possible to keep all the necessary resource attributes in the TCB entry, so that each request only operates within its own TCB entry. There is no need for any critical section as each TCB entry handle can be used throughout. The only cost is an increase in TCB entry size from 120 bytes to 320 bytes. This approach simplified our programs and resulted in a scalable design for Web servers.

2.6 Minimal resource dependence

In a bare PC, it is crucial to reduce resource dependencies in order to design a simple low-cost system. In case of the Web server application, this is easy to do since it does not require complex interfaces. The HTTP resource files are read from a network and stored in memory for fast access during execution. This reduces I/O dependencies and simplifies the scheduling structure.

As memory is now abundant and cheap, there is no need to conserve its usage. A bare PC Web server uses real memory to avoid paging and other overhead. A floppy interface is used to boot and load an AO, because this is the simplest interface to start and run the system. We avoided writing a graphics device driver as it is not essential in a Web server.

2.7 Single address space

The Web server AO is a single monolithic executable. It contains all the necessary code and hardware interfaces provided as a C++ API. There are no system calls or system libraries needed to run the Web server application. The system also runs in one mode (i.e., user mode). This design results in simplicity and gives system control directly to the user avoiding the need for middleware or other layers.

3. Design and Implementation

3.1 Alternate Designs

Three different designs for a bare PC Web server are discussed in this paper. As shown in Figure 1, the Web server architecture is implemented using C++ and a direct API to hardware, which is encapsulated with the application program. The Ethernet, IP, TCP and HTTP protocols are implemented without adhering to strict layering conventions so that any layer can communicate with any other layer. The Web server can operate with an internal chip or external NIC. Their respective drivers are encapsulated into the Web server application program.

The first design used an external NIC and multi-tasking, and is referred to as the EMT (External MultiTasking) server. It incorporates many of the features discussed in Section 2 such as spawning a separate task that runs to completion for each HTTP request.

The Web server based on the second design, which assumes an internal (onboard) NIC, is referred to as the IMT (Internal MultiTasking server). The IMT server is similar to the EMT server except that the device driver for the NIC is based on the internal Intel chip.

The third design, which also assumes an internal NIC, includes a novel task structure that relies on a single HTTP task to process all requests thus avoiding task switching overhead. When this task runs, it processes all requests that are waiting to be served and continues processing all the requests in the run list before returning to the Main task. If a new request arrives while this task is processing current HTTP requests, the new request will wait until the HTTP task returns to the Main task. The Web server based on this design is referred to as the IST (Internal Single Task server). In essence, the IMT server has multiple tasks for multiple requests, whereas the IST server has a single task for multiple requests. Thus, the IST design avoids creating multiple tasks, management of multiple tasks, and task related overhead.

3.2 Hardware API

The hardware API [19] consists of interfaces to keyboard, display, tasks, floppy drive, and NIC. There are also interfaces for boot, load, memory dump, memory search, debug, and trace that are not described here. Examples of the C++ API calls for the keyboard include AOAgetCharacter(), AOAgetCharacterBuffer(), AOAgetDecimal(), AOAgetHex(), and AOAgetString(). These interfaces allow data to be read from the keyboard either synchronously or asynchronously. An interface task is used to invoke these methods directly from a C++ program to avoid synchronous waiting in the program, as this task can be suspended while waiting for the keyboard data. Software interrupts are used to implement these interfaces using assembly language. This code is written as a hardware API and an application programmer has no need to understand or modify this code.

The examples of display interfaces include AOAprintHex(), AOAprintDecimal(), AOAprintCharacter(), and AOAprintText(). These interfaces are implemented in assembly language and use the video buffer for controlling the screen display. There are many other cursor interfaces to manage the screen output.

The task interface examples for the Intel 386 (or above) CPUs include AOAcreateTask(), AOAinitTask(), AOAsuspendTask(), AOAswapTask(), AOAinsertTask(), and AOAcompleteTask(). These interfaces are implemented in a single class, where the internal implementation uses assembly code to control and manage task segments as well as global and local descriptor tables. An AO programmer simply uses the above interfaces in a C++ program and does not need to have any knowledge of the Intel CPU architecture. The programmer can define a single C++ class with task methods, where each method can be defined as a separate task that can be managed using the above functions. In our experience, we have found that programmers writing bare PC applications are able to successfully use the hardware API without having any knowledge of assembly code or the internal architecture of the Intel CPU.

The file interface examples include AOAreadFromFloppy() and AOAwriteToFloppy() for communicating with the floppy drive. These interfaces use INT 13H internally to access the floppy drive using call gates in the Intel CPU architecture. The call gate code written in assembly is hidden in the interface. We have also written similar interfaces for a flash drive (these can be directly invoked from a C++ program).

As described above, the direct API to hardware is simple and easy to use. The Web server only encapsulates the needed interfaces in its AO. An AO programmer can use these interfaces in the C++ program without any knowledge of the machine architecture. As DOSC is based on an application-oriented approach, it only carries the necessary operating environment thus making it simple, robust, and extensible. It only carries functionality essential for the given application. Since the basic design of the bare Web server is based on the DOSC paradigm, it can be conveniently extended to incorporate graphics, security, or other desired functionality. This is possible because the AO is not dependent on any OS environment.

3.3 Code Size

Most of the code for a bare PC server is written in C++; the API for hardware and the Ethernet drivers involves Microsoft assembler code. The development is done in the DOS window environment using batch files so that no system libraries are included in the executable. Code size as well as executable size for each server is not significantly different. For example, the IMT server has about 8751 executable statements including the C++ code for the hardware interfaces. The external NIC driver has about 1506 lines of assembly code including comment lines; the internal NIC driver has 26 executable statements of assembly code (most of its code is written in C and C++). The C++ API to hardware has about 1656 lines of assembly code including comments (much of this API is used for printing, testing, and debugging the code). Executable size is 139,264 bytes not including the boot code and loader. All executables including boot and load modules fit on a single floppy disk so that the Web server is portable and easily run on older PCs (or PCs that are discarded due to upgrades) anywhere in the world with no OS requirement. This could significantly reduce the cost of deploying Web servers (and clusters of Web servers) in areas of the world where the cost of buying newer PCs and high-end server machines is prohibitive.

The bare PC Web server currently supports requests for static files only. It hosts HTML files (there is no limit to the number of files other than the PC's 512 MB memory limit, which can be expanded to 2 GB if needed). In general, as a bare PC Web server uses real memory, its memory capacity can be increased to any desired level. For a single CPU system, memory is not an issue when designing a single Web server (for a cluster of servers, the memory available in multiple PCs provides the required capacity). Conventional Web servers such as Apache and IIS use hard disk and cache memory to decrease memory latency. They incur overhead in paging and virtual memory translations that is avoided in the bare PC Web server.

3.4 Real World Applications

The approach used to build the bare PC Web server has already been used to build other bare PC real world applications. For example, we have built a VoIP softphone [23] and an email client that run on a bare PC. These applications have been tested and found to perform successfully in both LAN and Internet environments. The bare PC Web server has been operational since 2005 on a 24/7 basis at http://dosc.towson.edu and receives numerous hits per day. A bare PC Web server has additional benefits compared to a conventional OS-based Web server. As the former is smaller in size, simpler in design and easier to extend, it can be used as a personal Web server without requiring any commercial software. It can also be conveniently carried by the user on a flash drive and deployed on any bare PC. Furthermore, the ability to host the server on any Intel 386 (or above) based PC helps to reduce the wastage due to discarding old machines.

4. Performance Measurements

There are many ways to measure Web server performance [3, 26, 33]. The performance studies we have conducted, and the results presented in this paper are intended to complement the preceding discussion concerning the potential advantages of a bare PC Web server. They suffice to demonstrate the significant performance benefits of the bare PC Web server architecture, design, and implementation. However, these studies should not be viewed as the focus of the paper.

The bare PC Web server used for these tests is a static HTML server with full functionality. Its performance was compared with that of a Microsoft IIS server running Microsoft Windows Server 2003 OS, and Apache 2.2.3 [2] and Tux 2.0 [34] servers running Red Hat Linux 8.0. To ensure fair comparison, we used the recommended settings for maximum performance on the conventional Web servers. We also disabled unnecessary services on these servers because when a simple action such as opening a new window, or moving a mouse was attempted, their performance dropped dramatically. All experiments used a dedicated 100 Mbps Ethernet LAN in which client PCs connect to the server via a Netgear FS 608.v2 switch. The tools used for workload generation were http_load-12mar2006 [16] and httperf-0.8 [15]. The latter was also used in the performance studies done in [33]. The PC used for running the servers is a Dell Optiplex GX260, 2.4GHZ, with 512MB memory (this is an ordinary PC with a single CPU, not a server machine as our focus is on any bare PC). Four client PCs with similar clock speeds and hardware configurations were used: GX 200 (A), GX110 (B), Optiplex GX1 (C), and Optiplex GXa (D). They all ran Red Hat Linux 2.4.20-8.

The http_load generator is able to run up to 1000 fetches (HTTP requests) per second. It provides measurements for first-response time, connection time, throughput in bytes (number of fetches * bytes transferred per fetch), and max parallel (the maximum number of connections at any given point). It also reports minimum, maximum, and average values for each measurement, and shows how many requests timed out or had no response. We compare performance measurements of the servers with respect to first-response time and connection time while varying the number of fetches or the file size. We also measured task utilization for the bare PC Web server and tested the different bare PC server designs and examined their tradeoffs. The key performance results are summarized in Table 1. We discuss the results in more detail in the following sections.

Figure 3 illustrates a typical client server message exchange during a single fetch, which consists of connection, data transfer, and termination phases. It also shows the points at which RTT measurements can be made.

4.1 First Response Time

These measurements used an image file containing 3593 bytes. Client PC A was used to generate 1000 fetches per second, and client PCs B, C, and D were added to increase the load. Figure 4 shows the first-response time for the servers. The response time for the bare PC server configuration used in this set of experiments, referred to as the IMT server (integrated NIC with multiple tasks for multiple requests), is about 3.55 times better than the IIS server, 3.57 times better than the Apache server and 2.41 times better than the Tux server at 2500 fetches. The IIS and Apache servers do not run without errors or timeouts after 2500 fetches. On the other hand, the Tux server and bare PC server can handle up to 2833 fetches. The limit for the number of fetches per second may be estimated as follows:

bandwidth = 100,000,000 bits/sec

filesize = 3593 bytes

hdrsize = 58 bytes (18 (ETH) + 20 (IP) + 20 (TCP))

numPackets = 7 (SYN-ACK, GET-ACK, Data (4), ACK)

maximum number of fetches/sec = bandwidth/((filesize+hdrsize*numPackets)*8) = 3126.

Thus, Tux and the bare PC server both achieve results close to the upper bound. However, the bare PC server has the lowest first response time due to its simple task management scheme and minimal request processing overhead. The server is designed so that each task only performs essential work and returns to Main task, ensuring efficient use of the CPU. In contrast, the OS-based servers use interrupt-service routines and special instructions to process client requests, but they still incur kernel level overhead and delay that are absent in a bare PC system.

Figure 5 compares the bare PC server response times with a varying number of fetches obtained for a 3593-byte file using the http_load and httperf tools workload generators. It can be seen that both generators yield similar results.

4.2 Connection Time

Figure 6 shows the average connection time for the servers. For 2500 fetches, the bare PC server connection time is 1.444 times better than the IIS server, 1.26 times better than Apache server and 1.237 times better than the Tux server. Note that the connection time at the client side is composed of the various RTTs shown in Figure 3. One of these is the delay that elapses between receiving the client’s ACK and the subsequent GET request at the server. This delay is plotted with respect to the number of fetches in Figure 7, and it varies between 0.2 and 0.8 milliseconds. This delay makes a significant contribution to the measured total connection time, and limits the performance improvement that can be achieved by the server.

Table 1. Summary of Web Server Performance

Measurement	**IMT**	**IST**	**Tux**	**IIS**	**Apache**
AverageResponseTime (ms) 1000 fetches FileSize 3593 bytes	0.168	0.182	0.569	0.536	0.662
AverageConnect Time (ms) 1000 fetches FileSize 3593 bytes	0.155	0.147	0.313	0.32	0.319
AverageResponseTime (ms) 143 fetches FileSize 79722 bytes	0.738	2.03	1.74	3.2	0.87
AverageConnect Time (ms) 143 fetches FileSize 79722 bytes	0.7	5.44	1.589	1.93	1.06
MaxNumberOf Fetches FileSize 3593 bytes	2833	2833	2833	2500	2500

4.3 File Size Variation

Figure 8 shows the average response time for each server when file sizes are varied from 3593 bytes to 79,722 bytes. For the largest "filesize" of 79,722 bytes, using a calculation similar to that given in Section 4.1, the maximum number of fetches is estimated to be 150. The bare PC server is able to handle up to 143 fetches, which is again close to the limit. Its average first-response time for a 36,016 byte file is 4.345 times better than the Apache server, 3.85 times better than Tux server, and 5.43 times better than the IIS server. Moreover, as seen in Figure 9, the bare PC server's average connection time is 1.88 times better than the Apache server, 1.86 times better than the Tux server, and 2.516 times better than the IIS server. For large file sizes, the Apache server has better results than the IIS and Tux servers. However, the bare PC server has better response time and connection time up to a file size of 60,000 bytes.

We also measured the average response time for a 79,722-byte file by varying number of fetches. The results, shown in Figure 10, indicate that for 143 fetches, the bare PC server outperforms the other servers and is more stable with respect to load variation. The efficiency of the bare PC server is due to its streamlined design. There is little delay in processing a request since all the packets to be sent are placed in the DPD buffer if the client window size is large enough. Subsequent packets are sent with minimum delay since the waiting task is activated as soon as an acknowledgement is received from the client. This resume action is instantaneous as the acknowledgement changes the state of the TCB entry which in turn starts the execution of a suspended HTTP task.

4.4 Utilization

Figure 11 shows utilization for the RCV and HTTP tasks with a varying number of HTTP requests and a file size of 3593 bytes. Recall that there are three types of tasks as shown in Figure 2: the RCV task, HTTP tasks (many) and the Main task. The results indicate that the RCV task always consumes more time than the HTTP tasks for small file sizes. This is because the RCV task handles a request all the way from UPD buffer till the TCP connection is established and the HTTP request is initiated. On the other hand, an HTTP task simply performs work to send data packets to the NIC (which only involves adding the header followed by DPD insertion); it then returns to the Main task for the duration of the RTT while waiting for a response. The percentages of time used by the RCV task, HTTP tasks, and Main task respectively are 23%, 15%, and 62% respectively. Note that the time used by the Main task is the idle time of the server. The results indicate that the bare PC server consumes minimal CPU time and is thus able to process more requests.

Figure 12 shows the task utilization percentages for a file size of 79,722 bytes and 143 fetches. The percentages of time used by the RCV, HTTP, and Main tasks are 8%, 16%, and 76% respectively. In this case, the RCV task utilization is less than the HTTP task utilization because there are few packets received and a large number of packets to be transmitted due to the large file size.

Figure 13 shows the maximum number of tasks run and maximum number of TCB entries used at any time for the bare PC server. The bare PC server only used a maximum of 17 tasks and a maximum of 25 TCB entries to handle requests. Figure 14 shows the values reported by http_load for "max parallel". We observed that these values are not consistent as a different value is obtained for the same run at different times.

4.5 Alternative Server Designs

Figure 15 shows the first response time for three alternative designs of the bare PC Web server. These measurements use a file of 3593 bytes, and the number of fetches varies from 100 to 1000 fetches. The results shown were obtained using a hub instead of a switch. Figure 16 shows the connection time for three designs. The bare PC Web server referred to as the EMT server has an external NIC (3com 905CX card). The servers referred to as the IMT and IST servers use an internal NIC (Intel chip 82540EM). We investigated three designs to compare their respective tradeoffs. As different PCs use different NIC chips on the motherboard, if we have two drivers (for the motherboard NIC and external NIC respectively), it is possible to run the bare Web server in any Intel 386 (or above) based PC.

The results indicate that the IMT and IST servers have almost identical performance up to 1000 fetches. Although the EMT server with an external NIC also has reasonable performance, an internal NIC on the motherboard provides better results. Figure 17 shows that when the hub was replaced by a switch, the IMT and IST server designs exhibited similar performance up to 2500 fetches, but the IMT server performs better after 2500 fetches. Figure 18 and Figure 19 show the first response time, and connection time of the IST and IMT designs for various file sizes. A single task handling multiple requests has a greater response time for file sizes larger than 50,000 bytes as it requires more time to transmit a larger file. In this case, one task also has to check the state of all other tasks to send files. This limits its ability to receive more requests from the client and results in an increased response time. Note that a single task serving multiple requests has comparable performance to multiple tasks serving multiple requests. This means that the simple IST design can be used under moderate load conditions without a significant loss in performance but with the benefit of a simpler task structure (Main task, RCV task, and one HTTP task) and simpler overall server design. Thus, it is possible to trade off simplicity versus complexity when designing a bare PC Web server.

5. Conclusion

This paper presents a novel approach to building bare PC Web servers using the dispersed operating system computing paradigm. A bare PC Web server inherently provides single copy buffering, avoids concurrency control, and minimizes interrupts. It also has certain intrinsic features in its design such as simplicity and smaller code size that are much harder to achieve in a commercial Web server.

Three variations of bare PC Web server designs are studied: a server with an external NIC, a server with an internal NIC and single task structure, and a server with an internal NIC and multi-tasking structure. It is shown that the multi-tasking server design performs better than the IIS, Apache and Tux servers, and also better than the single tasking server when the number of fetches exceeds 2000. The multi-tasking server and the Tux server can handle up to 2833 fetches, which is close to the upper bound for a 100 Mbps network. In contrast, the IIS server and Apache server can only handle up to 2500 fetches. The response time and connection times for the multi-tasking server are much better than those for the other servers. The single task server design under normal load conditions also performs better than the IIS, Apache, and Tux servers.

We are presently developing a bare PC Web server to handle dynamic content. We are also investigating bare PC Web server scalability, including the use of bare PC Web server clusters. A topic for future research is that of security issues relevant to bare PC Web servers and bare PC applications in general.

Acknowledgements

We sincerely thank NSF for partial support of this work. We also express our gratitude to the late Dr. Frank Anger, who initially supported bare PC computing work by funding SGER grant CCR-0120155. Finally, we thank 3Com for giving us permission to use their programmer reference manual to build the device driver for the 3C905CX Ethernet network interface card.

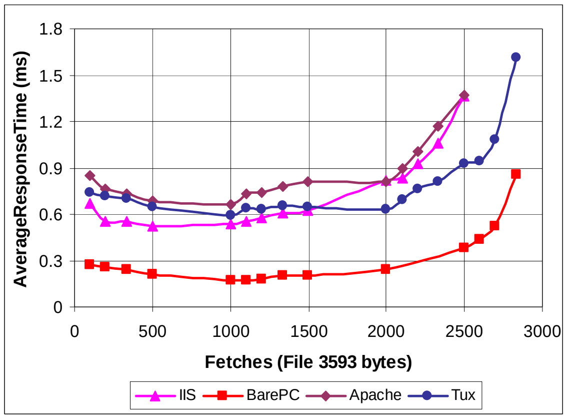

Figure 4. Average Response Time versus Fetches

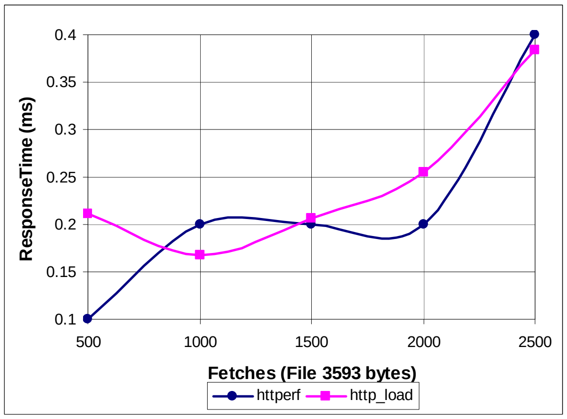

Figure 5. http_load and httperf Comparison

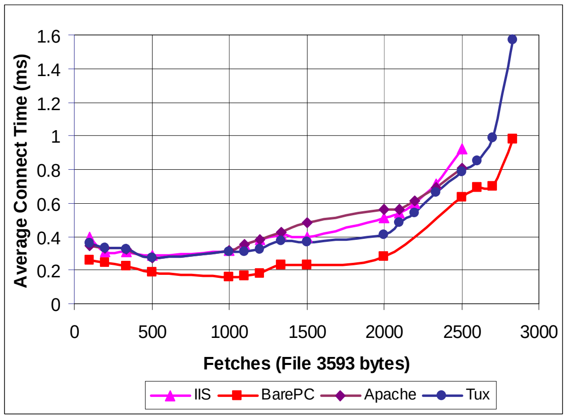

Figure 6. Average Connection Time versus Fetches

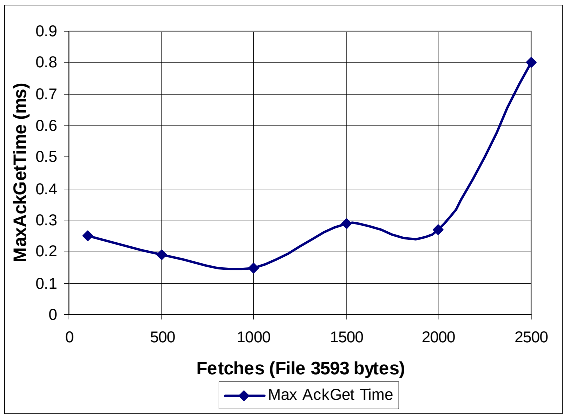

Figure 7. Maximum ACK-GET Time

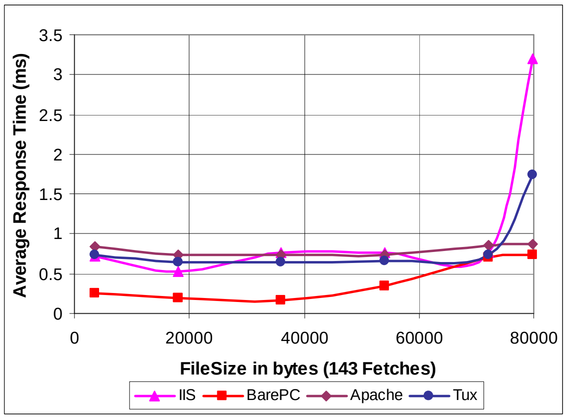

Figure 8. File Size versus Response Time

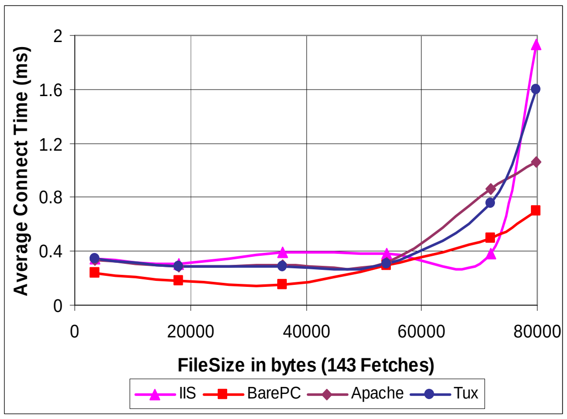

Figure 9. File Size versus Connect Time

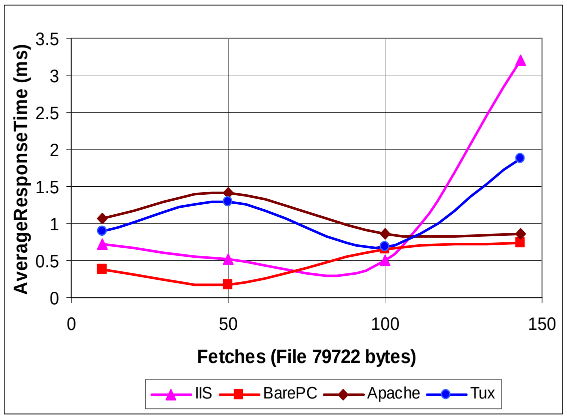

Figure 10. Response Time vs Fetches for Large File Size

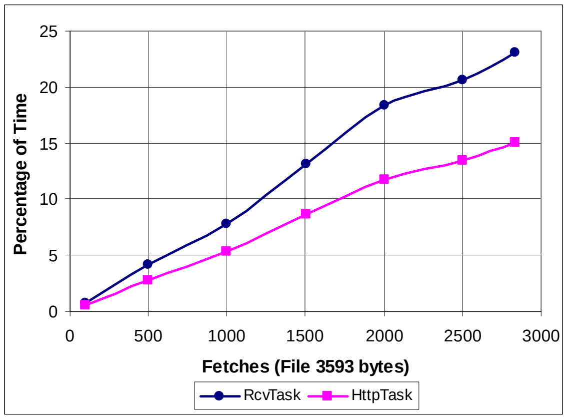

Figure 11. Task Utilization

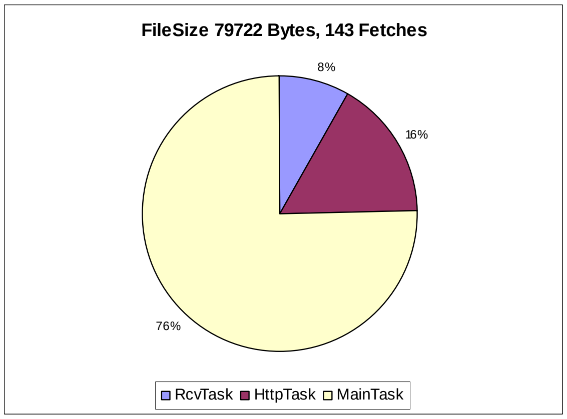

Figure 12. Task Utilization

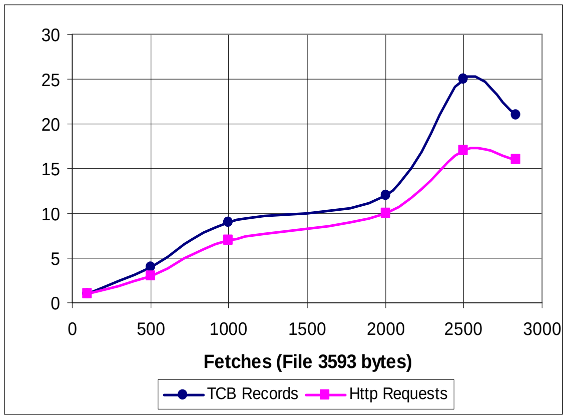

Figure 13. TCBMax & HttpMax versus Fetches

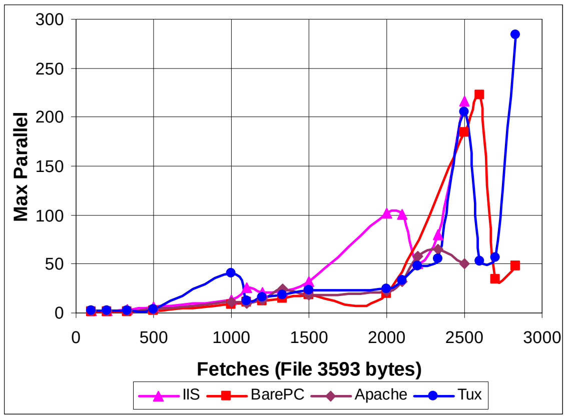

Figure 14. Max Parallel versus Fetches

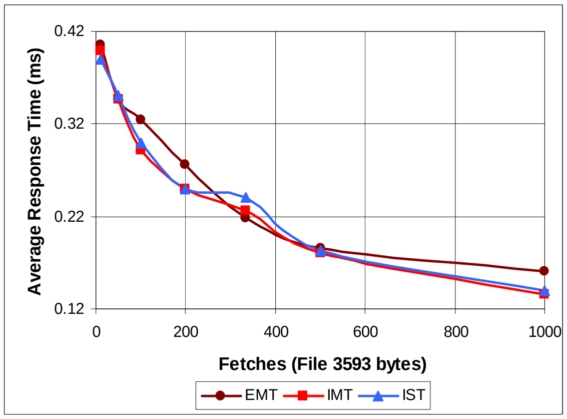

Figure 15. Response Time versus Fetches for various Server Designs

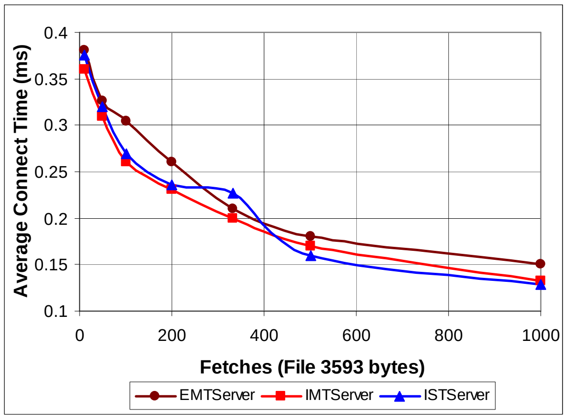

Figure 16. Connect Time versus Fetches for various Server Designs

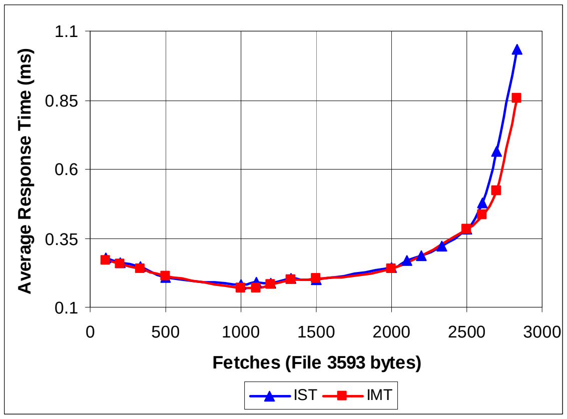

Figure 17. Response Time versus Fetches for IST and IMT Server Designs

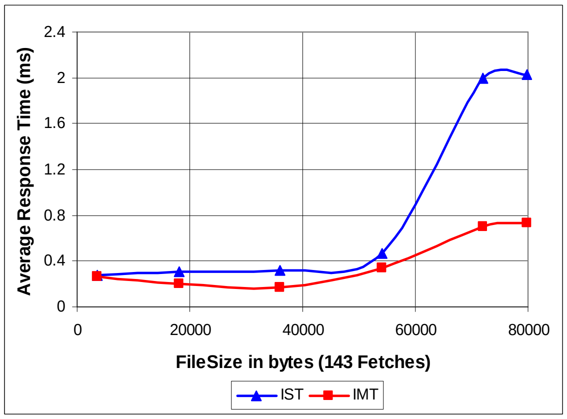

Figure 18. Response Time versus File Size for ISTand IMT Server Designs

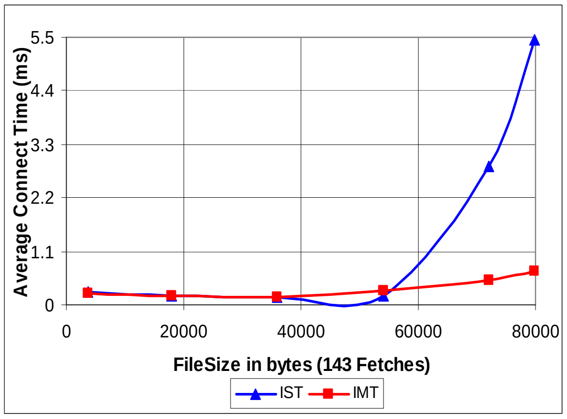

Figure 19. Connect Time versus File Size for ISTand IMT Server Designs

References

[1] M. Andersson, J. H. Cao, M. Kihl and C. Nyberg. "Performance Modeling of an Apache Web Server with Bursty Arrival Traffic." In *International Conference on Internet Computing (IC)*, June 2003.

[2] "The Apache Web server." http://www. apache.org/.

[3] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm and R. Katz. “TCP behavior of a busy Web server: analysis and improvement.” In *Proceedings of IEEE INFOCOM, San Francisco, CA*, 1998.

[4] L. A. Barroso, J. Dean and U. Holzle. “Web Search for a Planet: The Google Cluster Architecture.” In *IEEE Computer Society* 2003.

[5] “Buffer Copy Elimination.” http://public.boulder.ibm.com/infocenter/tivihelp

[6] T. Brecht, D. Pariag and L. Gammo. “Accept()able Strategies for Improving Web Server Performance.” In *Proceedings* of *USENIX 2004 Annual Technical Conference*, General Track Pp. 227–240, 2004.

[7] Y. Diao, J. L. Hellerstein, S. Parekh and J. P. Bigus. “Managing Web Server Performance with Autotune Agents.” *IBM Systems Journal*, Vol 42, No.1, 2003.

[8] K. M. El Defrawy, M. S. El Zarki and M. M. Khairy. “Proposal for a cross-layer coordination framework for next generation wireless systems.” In *Proceedings of the 2006 International Conference on Communications and Mobile Computing IWCMC '06*, 2006.

[9] D. R. Engler. “The Exokernel Operating System Architecuture.” Ph.D. thesis, MIT, October 1998.

[10] D. R. Engler and M.F. Kaashoek. “Exterminate all operating system abstractions.” In *Fifth Workshop on Hot Topics in Operating Systems*, p. 78, 1995.

[11] B. Ford, M. Hibler, J. Lepreau, R. McGrath and P. Tullman. “Interface and execution models in the Fluke Kernel.” In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 101-115, 1999.

[12] L. Gauthier, S. Yoo and A. A. Jerraya. “Automatic generation and targeting of application-specific operating systems and embedded systems software.” In *Computer-Aided Design of*

Integrated Circuits and Systems, IEEE Transaction, volume:20, Issue:11 On page(s): 1293-1301, 2001.

[13] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt and T. Pinckney. "Fast and flexible application-level networking on exokernel system." In *Proceedings of ACM Transactions on Computer Systems (TOCS)*, Volume 20, Issue 1, Pages: 49 – 83, February, 2002.

[14] L. He, R. K. Karne, A. L. Wijesinha, S. Girumala and G. H. Khaksari. "Design Issues in a Bare PC Web Server." In *Proceedings SNPD 2006*, pp 165-170, 2006.

[15] Httpperf. http://www.dc.fit.qut.edu.au/cgi-bin/tar.cgi

[16] http_load. http://www.acme.com/software/http_load

[17] M. F. Kaashoek, R. Elngler, G. R. Ganger and D. A. Wallach. "Server Operating Systems." In *1996 SIGOPS European Workshop*, pp. 141-148, September, 1996.

[18] R. K. Karne, K. A. Venkatasamy and T. Ahmed. "Dispersed Operating System Computing (DOSC)." In *Onward Track, OOPSLA 2005, San Diego, CA*, October 2005.

[19] R. K. Karne, K. Venkatasamy and T. Ahmed. "How to run C++ applications on a bare PC." In *Proceedings of 6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel / Distributed Computing (SNPD)*, May 2005.

[20] R. K. Karne. "Application-oriented Object Architecture: A Revolutionary Approach." In *6th International Conference, HPC Asia 2002*, December 2002.

[21] R. K. Karne, R. Gattu, R. Dandu, and Z. Zhang. "Application-oriented Object Architecture: Concepts and Approach." In *26th Annual International Computer Software and Applications Conference, IASTED International Conference, Tsukuba, Japan, NPDPA 2002*, October 2002.

[22] R. K. Karne. “Object-oriented Computer Architectures for New Generation of Applications.” In *Computer Architecture News*, Vol. 23, No. 5, pp. 8-19, December 1995.

[23] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He and S. Girumala. “A Peer-to-Peer Bare PC VoIP Application.” In *Proceedings IEEE Consumer and Communications and Networking Conference,* Las Vegas, Nevada, January 2007.

[24] H. Kim, V. Pai, and S. Rixner. “Increasing web server throughput with network interface data caching.” In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems San Jose, California,* October 2002.

[25] S. Lampoudi and D.M. Beazley. “SWILL: A Simple Embedded Web Server Library.” In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, Monterey, California,* June, 2002.

[26] G. Banga and P. Druschel. “Measuring the Capacity of a Web Server.” In *Proceedings USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, Dec. 1997.

[27] E. Nahum, T. Barzilai and D. D. Kandlur. “Performance Issues in WWW Servers.” In *Proceedings of IEEE/ACM Transactions on Networking*,, Vol.10, No.1, February 2002.

[28] V. S. Pai, P. Druschel and W. Zwaenepoel. “IO-Lite: A Unified I/O Buffering and Caching System.” In *Proceedings of ACM Transactions on Computer Systems (TOCS*), Volume 18, Issue 1, Pages: 37 – 66. February 2000.

[29] V. S. Pai, P. Druschel and W. Zwaenepoel. “Flash: An efficient and portable web server.” In *USENIX 1999 Annual Technical Conference*, pages 199–212, Monterey, CA, June 1999.

[30] M. P. Papazoglou and D. Georgakopoulos. “Service-Oriented Computing.” In *Communications of the ACM*, Vol.46, No.10, pp.25-28, October 2003.

[31] “The OS Kit Project.” http://www.cs.utah.edu/flux/oskit.

[32] "TinyOS is an open-source operating system designed for wireless embedded sensor networks." http://www.tinyos.net/.

[33] L. Titchkosky, M. Arlitt and C. Williamson. "A Performance Comparison of Dynamic Web Technologies." In *ACM SIGMETRICS Performance Evaluation Review,* volume 31, 2003.

[34] "Tux Server." http://www.stllinux.org/meeting_notes/2001/0719/tux/intro.html/.

[35] T. Venton, M. Miller, R. Kalla, and A. Blanchard. "A Linux-based tool for hardware bring up, Linux development, and manufacturing." *IBM SYSTEMS Journal*, Vol. 44, No. 2, p319-329, 2005.