Memory Access Characteristics of Network Infrastructure Applications

Abdul Waheed
Computer Engineering Department
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
E-mail: awaheed@ccse.kfupm.edu.sa

Abstract

Network infrastructure is composed of various devices located either in the core or at the edges of a wide-area network. These devices are required to deliver high transaction throughput where a transaction may involve processing one protocol data unit (PDU). Throughput of network infrastructure applications running on general-purpose architecture based servers is constrained due to excessive memory access latencies and limited memory transfer bandwidth. In this paper, we analyze the memory access characteristics of three network infrastructure applications: IP forwarding, HTTP proxying, and RTP streaming. In addition, we analyze the latencies of these network applications with respect to three types of data transfers: memory-to-CPU, memory-to-memory, and memory-to-network. We calculate the optimistic upper-bounds on throughput for these applications on general-purpose computing platforms.

Keywords: Memory performance, memory access characteristics, network infrastructure applications, IP forwarding, HTTP proxying, and RTP streaming.

1 Introduction and Motivation

A wide-area network infrastructure generally consists of two types of devices depending on their location within such a network: edge devices and core devices. Typical core devices include switches and routers while edge devices include bridges, gateways, layer four switches, proxy servers, server accelerators, web servers, and streaming servers. All of these devices are required to deliver high throughput for network transactions that they process. The nature of these transactions differs from one network infrastructure device to another. For instance, a transaction at a router involves accepting an incoming IP packet, extracting its destination IP address, determining an output port by searching through its routing table, updating the IP header, and finally forwarding the packet to an output port. However, a transaction at a proxy server may involve accepting an HTTP request from a client, searching for the required document in its local storage, and delivering that document either from local storage or fetching it from the original web server and then delivering it to the end-user. Although these two are different types of transactions, there are some important similarities among these and other network infrastructure applications: (1) each network transaction is independent of the other; (2) both applications demand high transaction throughput; and (3) large volumes of protocol data units (PDUs) transfer in and out of the system with minimal computation requirements. From an application perspective, data transfers among CPU, network, and I/O devices are essentially memory accesses. Therefore, high throughput in terms of transaction rate as well as data transfer in and out of a network device is dependent on memory access characteristics of the application and overheads introduced by the architecture while transferring large volumes of PDUs.

Traditionally network infrastructure is built using special-purpose architectures that help achieve high transaction throughput for specific applications. However, with exponentially increasing performance of general-purpose processors at comparatively low costs, there is renewed interest in utilizing general-purpose computing platforms for building network infrastructure [8]. Also, network infrastructure is increasingly becoming sophisticated and provides several value-added services in addition to simple switching and routing. Some of these services include overlay networking [3], content networking [2], layer four switching [1], and proxy caching [9]. Unlike simple routing and switching, these services require many more CPU cycles per transaction. As performance (in terms of clock speeds) of general-purpose processors continue to double roughly every eighteen months, using them to build value-added network infrastructure is becoming increasingly feasible. A number of commercial network infrastructure devices are implemented on general-purpose server platforms. For instance, Microsoft Proxy Server [12] is implemented on Windows 2000 Server platforms and can be used as an edge device. Similarly, Lucent Layer 4/7 switch architecture [1] uses general-purpose Unix/PC based platform.

Performance of general-purpose computing platform based network infrastructure is constrained by the discrepancy between processor and memory performance. While processor speed is increasing exponentially, the memory access latency is not reducing at the same rate. This discrepancy makes it exceedingly hard for an application developer to keep the processor busy during every clock cycle to realize its full performance potential. Since general-purpose computer architecture is based on multiple levels of memory hierarchy, it is not possible to keep the processor busy unless all memory accesses hit either in CPU registers or on-chip caches. Limited bandwidth of system and I/O buses further constrain the maximum data throughput possible from these architectures. In addition to these memory performance constraints, achieving peak processor performance becomes even harder due to system software overheads. Thus using a general-purpose processor based server for building network infrastructure devices has great potential in terms of cost-effectiveness and enabling intelligent network services while there are significant technical challenges that may inhibit the possibility of achieving high throughput. In this paper, we specifically analyze the severity of the first two challenges for network applications: cache performance overhead and overhead of data movement across system and I/O buses. Although the analysis of software overhead is very important for obtaining high throughput, it is beyond the scope of this paper to allow ourselves to focus on memory access characteristics.

Memory performance characterization and evaluation is traditionally related to designing new processor architectures. Trace-driven simulation technique is widely used for experimenting with various types of cache and memory architectures. With the advent of on-chip performance counters, application developers can also measure cache and memory performance of their software. But increasing complexity of processor architectures, in terms of out-of-order execution and overlapping computation with memory stall cycles to hide memory access latency, complicates the use of low-level measurements for memory performance analysis or tuning of a software application. Although generally this techniques is unattractive, some memory performance measurement based analysis and tuning studies have be conducted on high-end systems for computational applications (see for example [11]). Sohoni et al. [10] analyze the characteristics of memory performance of streaming media applications. Their measurement and trace-driven simulation based study of streaming media player applications indicate that the cache miss ratio for such applications is lower than the cache miss ratio of computational applications in SPEC benchmark suite. This is due to streaming media algorithms that access contiguous blocks of data resulting in high spatial locality despite poor temporal locality. Kumann et al. [6] describe a benchmark that can be used for sequential and parallel systems for measuring data transfer bandwidth. In order to avoid the limitations and inaccuracies of measurement and trace-driven simulation techniques, we use simple analytical calculations in accordance with the memory access characteristics of three selected network infrastructure applications: IP packet forwarding, HTTP proxying, and Real Time Protocol (RTP) streaming. Using this technique, we first calculate best, worst, and average cache overhead in terms of memory stall cycles that may result for the applications of our choice. Subsequently, we calculate the latencies for a transaction for each of these applications in terms of number of data movements and bandwidths of internal and external buses. These calculations can be further simplified to estimate the optimistic upper-bounds on throughput for the above three applications on various general-purpose processor based architectures.

In Section 2, we outline our methodology to analyze memory accesses for network applications on a general-purpose computing platform. We apply this methodology to three selected network infrastructure applications in Section 3. We conclude in Section 4 with a discussion of the contributions of this work and future course of this research effort.

2 Analysis of Memory Access Latencies

Figure 1 illustrates a typical general-purpose architecture that can be used for high-throughput network infrastructure devices. 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. These hardware resources are connected to one another through a higher-bandwidth internal system bus, a lower-bandwidth I/O bus, and a bus controller. In term of data flow, both within as well as outside the server, there are four data transfer paths. These four 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 the CPU) for copying blocks of data from one network protocol layer to another; (3) disk-memory data transfers through 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 second types of data transfers use internal bus, the third and fourth types of transfers utilize I/O bus.

Transactions performed by a typical network infrastructure server can be characterized by three activities: (1) reading an incoming transaction request from a memory location (network buffer) through a network interface; (2) request processing that require 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

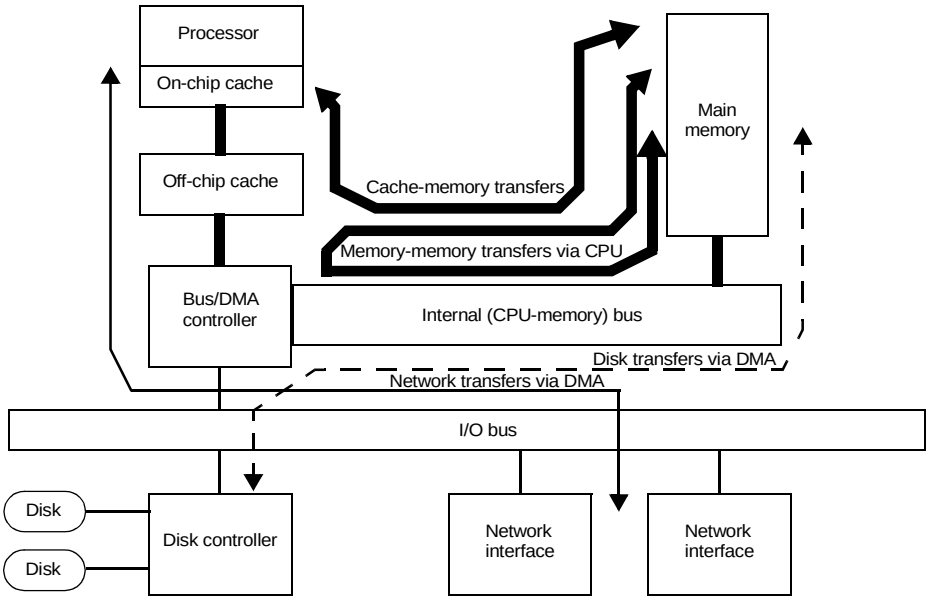

Figure 1: Architecture of a typical server built on a general-purpose platform with four data transfer paths.

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. Our objective in the rest of this section is to determine latencies due to each of these memory access operations.

2.1 Memory-CPU Transfers

Parts of a network transaction utilize 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 of these functions require updating a protocol header, which consists of a typically small number of bytes. However, some operations such as checksum calculation of an entire 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 [5], such that:

$$\text{Memory stall cycles} = (IC)\,(AR)\,(MR)\,(MP) \qquad (1)$$

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:

$$\text{Memory stall cycles} = (IC)\,(MR)\,(MP) \qquad (2)$$

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 8 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 $MR = 1/8$ or 12.5% in this case. Memory stall cycles will be very close to the instruction count in such a case. Generally, $MR = 1/(L/W) = W/L$ where *W* is the width of each memory access (in bytes) and *L* is the length of each cache line (also in terms of 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:

$$(\text{Execution time})_{\text{no-cache}} = (IC)\,(CPI)\,(CC), \quad (3)$$

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:

$$(\text{Execution time})_{\text{with-cache}} = (IC)\,(CPI)\,(CC)\,\{1 + (MR)\,(MP)\} \quad (4)$$

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

$$\text{Cache overhead} = 1 + (MR)\,(MP) = 1 + (10)\,(\text{MR}) \quad (5)$$

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.

2.2 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:

$$\text{Memory-memory latency} = 2S/B_i\ \mu\text{sec}. \quad (6)$$

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 a minimum of 4 GBytes/sec of data transfer over its internal bus. This is equivalent to 32 Gbits/sec of data transfer rates within the server. Theoretically, such data transfer speeds can allow to build a software router that can provide more than 30 independent channels, each of 1 Gbits/sec capacity.

2.3 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:

$$\text{Memory-network latency} = S/B_e\ \mu\text{sec}. \quad (7)$$

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

3 Characteristics of Selected Applications

We apply the memory access latency calculation methodology developed in Section 2 to three network applications: IP forwarding, HTTP proxying, and RTP streaming. We assume that these applications are executed on a general-purpose processor based server, which is depicted in Figure 1. In each case, our goal is to identify the frequency of each of the three types of data transfer operations for every transaction. We assume that the latency of transaction is a sum of following latencies: processing by CPU, memory-CPU transfer, memory-memory copy, memory-NIC transfer, and memory-I/O transfer.

3.1 IP Packet Forwarding

A router is typically used in network core to perform two functions: (1) execution of a routing algorithm to periodically update its routing table; and (2) forwarding an incoming IP packet to a selected output port according to the information provided by its routing table. Execution of the routing algorithm is an infrequent operation compared to forwarding of IP packets, which should ideally be implemented at a level of throughput that matches the data rate of physical medium. Therefore, we focus on the IP packet forwarding part rather than routing function to identify the types and frequencies of data transfer operation that may limit the throughput.

An IP packet forwarding transaction involves 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 desti-

nation 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, this 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 result 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:

$$T_{IP} = T_{CPU1} + 2S/B_e \quad (8)$$

where T_{CPU1} is the CPU time taken by the transaction in microseconds and S is the size of PDU in bytes.

3.2 HTTP Proxying

Compared to an IP packet forwarding server, an HTTP proxy server can be much more complicated in terms of possible variations for each of its transactions. An HTTP proxy server is typically located at the edge of a network between a client and a number of web servers. An HTTP transaction from the user is intercepted by the proxy, which either delivers the required document from its local storage or contacts the origin server to get required document, stores it locally, and delivers it to the requesting client. In order to improve the response time, proxy server can store frequently accessed documents in main memory rather than disk. The decision of retrieving the document from local storage or origin server depends on the characteristics of the document accesses (popularity) as well as cacheability characteristics of a document determined, for instance, by their expiry dates. For many practical cases, the document hit rate in local storage is usually around 50% to 60% of all requested documents. In addition, the response depends on the type of request. For example, an HTTP GET request returns a document. However, an IF-MODIFIED-SINCE (IMS) may return the actual document if it were modified after the specified time or it may simply return NotModified status when the cached document is up-to-date. Throughput is maximized when all requested documents are hit in local storage in the main memory and all transactions are HTTP GET requests. We assume these conditions for subsequent analysis as our goal is to achieve maximum throughput from the server.

A typical HTTP proxy transaction 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 following operations:

1. Preparation of sending 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.

These operation 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 HTTP proxy transactions that need highest throughput is given as:

$$T_{HTTP} = T_{CPU2} + 5S/B_i + S/B_e \quad (9)$$

where T_{CPU2} is the CPU time taken by the HTTP transaction response in microseconds and S is the size of PDU in bytes.

3.3 RTP Streaming

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 readily 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 of these 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 frame. This RTP packet is copied to the transport layer (often using UDP rather than TCP). This involves one memory-memory data transfer.
2. Transport PDU is copied to an IP buffer. This operation involves one memory-memory data transfer of entire PDU.
3. Finally, the IP packet is handed over to the NIC resulting in on 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:

$$T_{RTP} = T_{CPU3} + 4S/B_i + S/B_e \quad (10)$$

where T_{CPU3} is the CPU time taken by the HTTP transaction response 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 three selected applications. Server throughput is given by S/T where S is the size of transaction data and T is the latency of a transaction given by equations (8), (9), and (10). Since we need to determine optimistic upper-bound on throughput for three selected 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:

$$(\text{Throughput})_{IP} = B_e/2 \quad (11)$$

$$(\text{Throughput})_{HTTP} = B_i B_e/(5B_e + B_i) \quad (12)$$

$$(\text{Throughput})_{RTP} = B_i B_e/(4B_e + B_i) \quad (13)$$

We can use these upper-bounds on throughput for several leading general-purpose microprocessors to calculate peak possible data throughput. These calculations are listed in Table 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 peak throughput estimates indicate that a general-purpose processor based server can deliver high throughput comparable to a server based on special-purpose architectures. For instance, a software router based on Intel Pentium IV processor can potentially provide at least 12 ports, each with a full-duplex data forwarding capacity of 155 Mbits/sec.

4 Conclusions and Discussion

This paper makes two important contributions: (1) using memory access characteristics of various network applications, we show that in most cases cache latency is typically less than twice the latency in ideal case that incurs no memory stall cycles; and (2) we provide a methodology of measuring throughput for three network applications and show that in each case throughput is limited by the internal and external bus bandwidths and number of data transfer operations required by each transaction. An important corollary of these conclusions is that, under the assumptions of no significant CPU overhead, no software overhead, and no contention for the buses, we can determine optimistic upper-bounds on throughput for selected network applications. Application of these results indicate that we can potentially use state-of-the-art general-purpose microprocessor based servers to run several network infrastructure applications and still meet their demands of high throughput. This conclusion is contrary to the common belief that special-purpose architectures with very small data caches are unavoidable for network applications where data re-use (temporal locality) is almost non-existent and caches hinder high data throughput rather than helping this process as in case

Table 1. Peak throughput of three network applications for leading general-purpose processors with different internal bus bandwidths. 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 selected network applications		
		IP forwarding (Mbits/sec)	HTTP proxying (Mbits/sec)	RTP streaming (Mbits/sec)
Intel Pentium IV 3.06 GHz	3200	4264	3199	3656
AMD Athlon XP 3000+	2700	4264	2867	3306
MIPS R16000 700 MHz	3200	4264	3199	3635
Sun Ultraspac III 900 MHz	1200	4264	1567	1873

of computational applications. If fact, as Sohoni et al. have noticed, cache miss rates for some network applications may even be higher than typical compute-intensive applications due to spatial locality of accessing blocks of contiguous data [10].

Although the peak performance estimates presented in Table 1 indicate that using a general-purpose processor based server for high-throughput network infrastructure devices is possible, there are some technical challenges that remain to be overcome. Eliminating excessive software overhead on a general-purpose computing platform is a significant challenge. Software overhead include: system call overhead, protocol processing (especially transport layer protocols that use sliding window for flow control), context switching overhead, thread management overhead, and contention handling of hardware resources. In addition, since a server is essentially a piece of software written in a high-level language, compiler plays an important role in obtaining high performance. If compiler generated object code is not optimized for a target architecture, it is impossible to achieve close to peak data transfer bandwidth from the system. Our analysis in this paper did not consider these system software and compilation related overheads. We are working on a modeling and measurement based approach to incorporate the impact of these software related overheads. A notable technique that has addressed the issue of software overheads for network applications is the Virtual Interface Architecture (VIA) [4], which allows for by-passing network layers as needed. However, this solution is limited to tightly-coupled cluster computing systems and may not work for a network server. Pradhan et al. [7] propose a cluster based IP router that uses multiple general-purpose computing platforms connected through a Myrinet switch to form a high throughput server. This solution does not address the problem of reducing overhead to achieve close to the peak performance that each general-purpose platform is capable of delivering. Instead, the solution results in high throughput only because processing work can be divided among multiple nodes, which may continue to operate sub-optimally, through a high-speed switch.

While the software overhead can inhibit potential high throughput from a network server implemented on a general-purpose platform, multithreading combined with superscalar execution can work in favor of a transaction processing application. Since all transactions are mutually independent, if one transaction results in stall cycles, CPU can switch to another thread that may be executable to process a different transaction. Multithreading can hide the latency due to memory access overhead and can offset the degradation due to memory stalls and software overheads.

Acknowledgements

Author would like to acknowledge the support of the Computer Engineering Department of King Fahd University of Petroleum and Minerals (KFUPM) for this work.

References

[1] Bell Laboratories, Lucent Technology, “Layer 4/7 Switch and other Custom IP Traffic Processing Using the NEPPI API,” *White Paper.* Available on-line from: http://www.bell-labs.com/project/webswitch/Gryph_im/APAH-BFEK.pdf.

[2] Yatin Chawathe, Steven McCanne, and Eric Brewer, “An Architecture for Internet Content Distribution as an Infrastructure Service,” *Technical Report,* University of California Berkeley, 2000.

[3] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang, “A Case for End System Multicast,” *in Proc. of Sigmetrics 20002*.

[4] D. Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke and Chris Dodd, “The Virtual Interface Architecture,” *IEEE Micro*, March/April 1998.

[5] John L. Hennessy and David A. Patterson, *Computer Architecture—A Quantitative Approach*, Morgan Kaufmann, Third Edition, 2003.

[6] 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,” *in Proc. of the 7th Workshop on Scalable Memory Multiprocessors held in conjunction with the 25th Annual International Symposium on Computer Architecture ISCA98*, Barcelona, Spain, June 27-28, 1998.

[7] Prashant Pradhan and Tzi-cker Chiueh, “Implementation and Evaluation of QoS-Capable Cluster-Based IP Routers,” *in Proc. of High Performance Networking and Computing Conference (SC’02)*, Baltimore, MD, Nov. 2002.

[8] Xiaoh Qie, Andy Bavier, Larry Peterson, and Scott Karlin, “Scheduling Computations on a Software-based Router,” *in Proc. of Sigmetrics 2001*.

[9] Dongjun Shin and Kern Koh, “Optimizing Web Content Delivery Using Web Server Accelerator,” *in Proc. of the Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002

[10] Sohum Sohoni, Rui Min, Zhiyon Xu, and Yiming Hu, “A Study of Memory System Performance of Multimedia Applications,” *in Proc. ACM SIGMETRICS 2001*, Cambridge, MA, 2001.

[11] M. Zagha, B. Larson, Steve Turner, Marty Itzkowitz, “Performance Analysis Using the Mips R10000 Performance Counters,” *in Proc. of Supercomputing ‘96*, Pittsburgh, Pennsylvania, Nov. 1996.

[12] Microsoft. http://www.microsoft.com.