White Paper

AltiVecTCPIPWP/D
Rev. 1, 1/2003

Enhanced TCP/IP Performance with AltiVec

Jacob Pan
CPD Applications

Abstract

Today's high speed networks are capable of delivering data at gigabit/second rates to desktop computers and some embedded systems. This dramatic improvement is taking place both in local area networks (LANs) where gigabit Ethernet is widely deployed and in wide area networks (WANs) where fiber optics is becoming dominant. The bottleneck in communications has shifted from the physical transmission media to the host processor, which typically runs a Transmission Control Protocol/Internet Protocol (TCP/IP) protocol stack to provide interconnection to most networking applications. In addition, the TCP/IP implementation presents contradictory requirements for processors to cope with both quick control branches and large data streams. This paper illustrates how performance of a TCP/IP protocol stack can be significantly improved by using AltiVec technology with little or no impact on the protocol source code.

Performance Enhancement

TCP/IP typically refers to the protocol suite that connects host computers through the internet. It encompass protocols such as Internet Protocol (IP), Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Internet Control Message Protocol (ICMP), Address Resolution Protocol (ARP), and File Transfer Protocol (FTP). This paper focuses mainly on enhancing TCP performance as an example in applying AltiVec technology in a real protocol stack implementation. TCP was chosen because it is the most complex protocol [1] in the suite and also because real network traffic shows that TCP packets comprise roughly 80% of all wide area network (WAN) traffic [1]. At the local area network (LAN) level, TCP packets also contribute to a significant portion of the network traffic but may not be dominant. This is mainly due to large amounts of non-IP carrying data link layer traffic. However, TCP also demonstrates strong locality between hosts, which implies that for some applications, TCP processing is still intensive at the LAN level.

In the following sections, an AltiVec-enabled software solution for TCP performance enhancement is revealed progressively, beginning with a bottleneck analysis of the protocol stack. Various solutions are then compared within stand-alone functions. In the end, the overall performance impact is presented with benchmark results. In addition, the process of identifying code fragments that can be vectorized is summarized; the same approach can be extended to other networking applications.

---

[1] TCP contributes to about 50 percent of the code in the entire TCP/IP stack.

Bottleneck Analysis in TCP/IP Protocol Stack

The implementation of a TCP/IP stack can vary greatly depending on hardware resources, memory footprint, and the type of data services required. Essentially, networking is about moving data from one location to another in a controlled fashion. To ensure either a reliable or a best effort data transfer service, the protocol stack needs to integrate fairly complex control logic to cope with the following:

- Packet differentiation
- Encapsulation/decapsulation
- Segmentation/reassembly
- Time-out and retransmission
- Acknowledgment
- Duplicate packet detection
- Tracking every byte of data with a sequence/acknowledge number
- Flow control
- Congestion avoidance

A complete implementation of the TCP/IP protocol stack often requires over ten thousand lines of C code. The code complexity is attributed mainly to the need for managing connection admission and dealing with network anomalies. However, in a well-designed network, the host processor spends much less time handling error conditions than performing normal data packet processing. But even the best implementation of a TCP/IP stack cannot avoid performing a data copy and a checksum operation at least once in each direction.

Therefore, the most common and expensive portion of the code is data manipulation, such as memory copy, memory initialization, and the checksum calculation. This characteristic is more significant with networking applications, such as FTP and SMTP, that use larger packet sizes. On the other hand, with the explosive growth of gigabit Ethernet, non-standard Ethernet jumbo packets (up to 9 Kbytes) are more and more attractive in some LAN environments. This is because larger frames usually mean fewer CPU interrupts and less processing overhead for a given data transfer size. Often the per-packet processing overhead sets the limit of TCP performance in the LAN environment.

The following equation explains how TCP throughput has an upper bound based on the following parameters:

$$\text{Throughput} \leq \sim 0.7 * \text{MSS} / (\text{RTT} * \text{sqrt}(\text{packet\_loss}))$$

where

RTT = round trip time

MSS = maximum segment size.

Therefore, maximum TCP throughput has a directly linear relationship to MSS; MSS is defined as the size of the maximum transfer unit (MTU) minus the number of bytes in the TCP/IP headers:

$$\text{MSS} = \text{MTU} - \text{TCP/IP headers}$$

To process a large standard TCP segment [1] or a jumbo frame, the number of control statements is independent from the block size. It may still involve processing a few ‘if’ statements, but they are essentially the same as for processing smaller packets. However, with a jumbo frame data manipulation requires more CPU clock cycles proportional to the packet size. For example, computing a TCP checksum is generally

[1] Up to 1460 bytes due to the limit of 1500 bytes payload in standard Ethernet.

considered to be one of the most expensive operations in the TCP/IP protocol stack because the checksum is based on both the header (both a TCP header and a pseudo IP header) and the entire TCP payload.

Figure 1 shows a typical checksum function in C.

```
unsigned short checksum(
                                unsigned short *addr, int len,
                                unsigned long sum, int byte_swap)
{
    unsigned short *pData = addr;
    while (len>1) {
        sum += *pData++;
        len-=2;
    }
    /* process odd byte */
    if (len==1) {
        if (byte_swap==0) {
            sum+=*(unsigned char *)pData;
        } else {
            sum+=(*(unsigned char *)pData)<<8;
        }
    }
    /*     add sum of carry bits back */
    sum=(sum>>16)+ (sum&0xffff);
    sum+=(sum>>16);
    return(~sum);
}
```

Figure 1. checksum Function

The code in the checksum function was compiled with a GNU C compiler with optimization level 2 and executed on a test bench with the configuration shown in Table 1.

Table 1. Test Configurations for Generic C checksum Function

Platform	Motorola Sandpoint X3
Processor	MPC7455
Core frequency	750 MHz
Bus speed	100 MHz
Test function	checksum
Compiler	GNU C 2.95 with -O2 option

In the best case (where data is resident in the data cache), this code can generate checksums at a rate of 3.07 clock cycles per byte, which is equivalent to 244 Mbytes/sec. In real TCP/IP processing. The worst case scenario occurs in the receive direction, where data packets are not cache-resident. This case was also experimented with by flushing the data cache each time before running the test. The result on the same MPC7455 processor shows that only 102 Mbytes/sec throughput can be achieved. Apparently, this performance poses a dramatic bottleneck to high-speed or bursty traffic such as gigabit Ethernet. Some embedded processors provide a built-in TCP/IP hardware assist unit to accelerate checksum computation; however, this approach requires extensive software changes to the existing stack and results in poor software portability.

In addition to computing the checksum, a protocol stack needs to move data in both receive and transmit directions between the application layer and physical devices. Packet segmentation and queueing also requires processing resources.

As a result, accelerating data manipulation/movement becomes a key factor to ease or eliminate this bottleneck in the TCP/IP stack. Later in this paper, the profiling result of a realistic TCP benchmark demonstrates the significance that a few frequently used data processing routines, such as memcpy(), checksum(), and memcpy_and_checksum() can have on performance.

Using AltiVec Technology in the TCP/IP Stack

The AltiVec technology in the MPC74xx processors is based on the implementation of separate vector/SIMD (single instruction stream, multiple data streams) execution units that have a high degree of data parallelism. A single instruction in AltiVec operates on multiple data items allowing for a faster and more efficient way to process large quantities of data. It also adds a 128-bit vector register file to the existing integer- and floating-point registers. The quad-word registers allow for quick processing on large amounts of data all at the same time.

Data manipulations in a TCP/IP stack are ideally suited for AltiVec instructions. For an example, a TCP checksum is computed as the 16-bit one's complement of the one's complement sum of all 16-bit words in the TCP segment header and TCP segment data. If the data does not end on a 16-bit boundary, it is padded with zeroes at the end. The C code implementation is shown in the code segment in Figure 1. If the checksum algorithm is coded with AltiVec instructions, the middle loop looks like the following:

```
vaddcuw  V_Carry_Current,VD1,VD2       // add data and store carries
vadduwm  V_Temp_Sum,VD1,VD2            // add data (no carries)
vaddcuw  V_Carry_Sum,V_Temp_Sum,V_Sum  // carry from sum update
vadduwm  V_Sum,V_Temp_Sum,V_Sum        // update sum
vadduwm  VCAR,V_Carry_Current,VCAR     // update carries from previous adds
vadduwm  VCAR,V_Carry_Sum,VCAR         // add carries from previous sum
```

Only six instructions are needed for computing a checksum for two quad words (32 bytes). Only one clock cycle is needed for each instruction on a MPC74xx processor. AltiVec also provides load and store instructions that can operate on 16 bytes at a time.

To speed up checksum and other data processing, most operating systems, such as Linux, implement these functions in scalar assembly code, which is optimized for each processor architecture. The most commonly used assembly functions used by TCP/IP stack include computing checksum, memory copy, and computing checksum in the memory copy loop.

Compared to scalar assembly instructions, computing the checksum or performing memcpy with AltiVec instructions not only requires far fewer logic/algebraic operations but also much less frequent use of the more time-consuming (expensive) load and store instructions. Also, the AltiVec instruction set allows efficient processing of various alignment cases.

In most cases, function interfaces are standardized and abstracted, so assembly functions can be linked directly with the protocol stack without changing the stack itself. The same convenient interfaces are also provided with AltiVec-enabled library functions. Therefore, with little porting effort, an existing TCP/IP stack can take advantage of the vector/SIMD engine that is available with the MPC74xx processor family.

To experiment with the performance of AltiVec-enabled library functions, the same hardware configuration as in Table 1 was used to compare the cases for the following:

- C/glibc

- Linux PPC Assembly
- AltiVec Assembly

Table 2 shows which figure provides the test case for the function being compared.

Table 2. Test Cases for Stand-Alone Functions

Function Name	Warm Cache	Cold Cache
checksum	Figure 2	Figure 3
memcpy	Figure 4	Figure 5
memcpy+checksum	Figure 6	Figure 7

It is important to notice that the AltiVec assembly functions used in this paper are not just optimized for limited scenarios, such as with a given alignment, but rather, they are written for general-purpose data processing [1]. For example, in this test all source and destination data buffers are only word aligned (aligned to 4-byte boundaries and not quad-word aligned [2]).

The following figures give side-by-side performance comparisons of each function. In addition, to measure real-world benefits, AltiVec-enabled library functions are tested with a TCP benchmark. Unless otherwise stated, the resulting data is all obtained from the same test bed described in Table 1.

Figure 2 shows the throughput comparison of the same checksum function shown in Figure 1 coded in AltiVec assembly, hand-coded scalar assembly from Linux [3], and the generic C function compiled by the GNU C compiler. The test repeats calling the checksum function 100 times on the same data buffer for each size, so that most of the computation can be assumed to be the warm cache case. To ensure fair comparisons, data caches are flushed each time before incrementing data sizes.

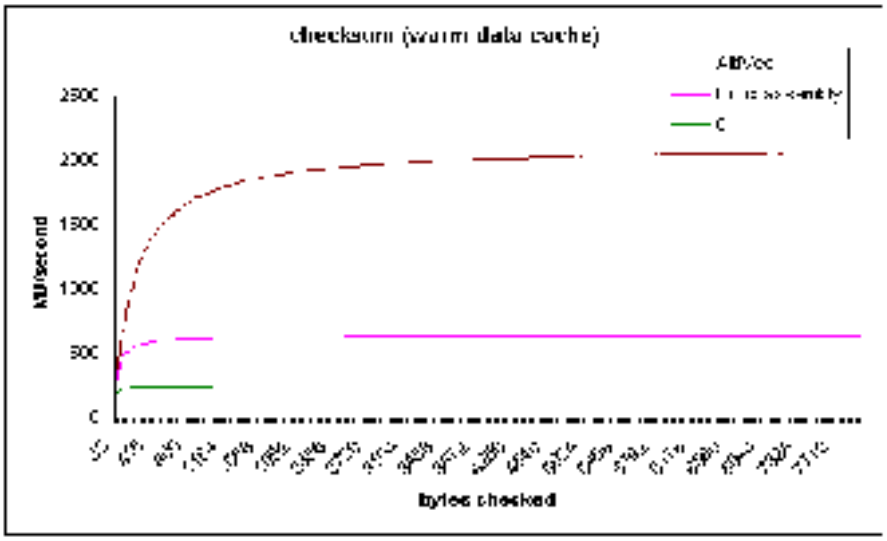

Figure 2. checksum Throughput with Warm Data Cache

[1] Performance of the above AltiVec assembly functions is fairly invariant to alignment.
[2] AltiVec load and store instructions operate on quad-word boundary.
[3] Linux 2.4.12 kernel

Minor differences are observed with small data packet sizes. However, as packet sizes increase, the AltiVec function provides increasing performance enhancement over scalar code. It can generate a checksum about four times faster than the hand-coded scalar assembly.

Figure 3 shows the same test in the worst case scenario. Bandwidth of the memory subsystem imposes a limit on this case. In most real world TCP/IP implementations, this scenario is typically avoided by computing a checksum in the same loop as memory copy. For example, in the receive direction, the checksum is computed while data is being copied from a receive ring buffer to a socket buffer. In the transmit direction, most data is obtained from the application layer via a socket buffer that is most likely cache-resident. Therefore, it is reasonable to assume that data is cache resident (or even register resident) while a checksum for TCP packets is computed.

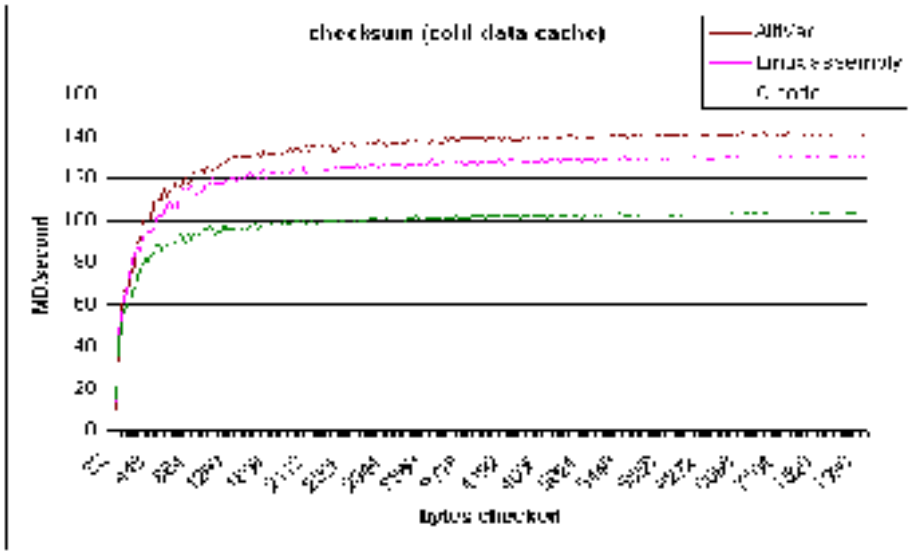

Figure 3. checksum Throughput with Cold Data Cache

The algorithm for the checksum examples who the increased performance even without the use of data streaming or large loop unrolling. These techniques, in addition to **dcbz** before stores, are used in the memcpy examples in Figure 4 and Figure 5.

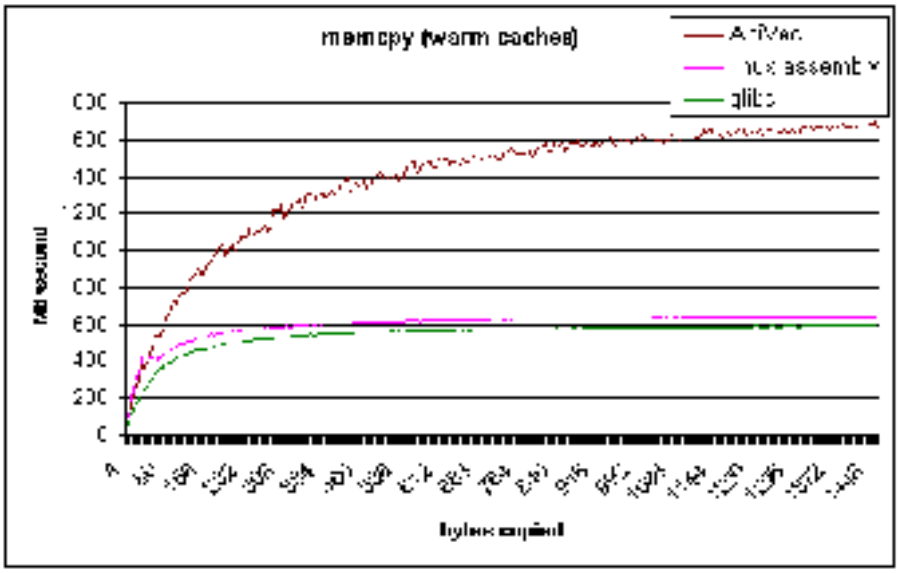

Figure 4. memcpy Throughout with Warm Data Cache

The warm cache example in Figure 4 yields an improvement similar to that of the warm cache case for checksum; however, using data streaming, loop unrolling, and **dcbz** before store techniques greatly improves the performance in the cold cache case, shown in Figure 5.

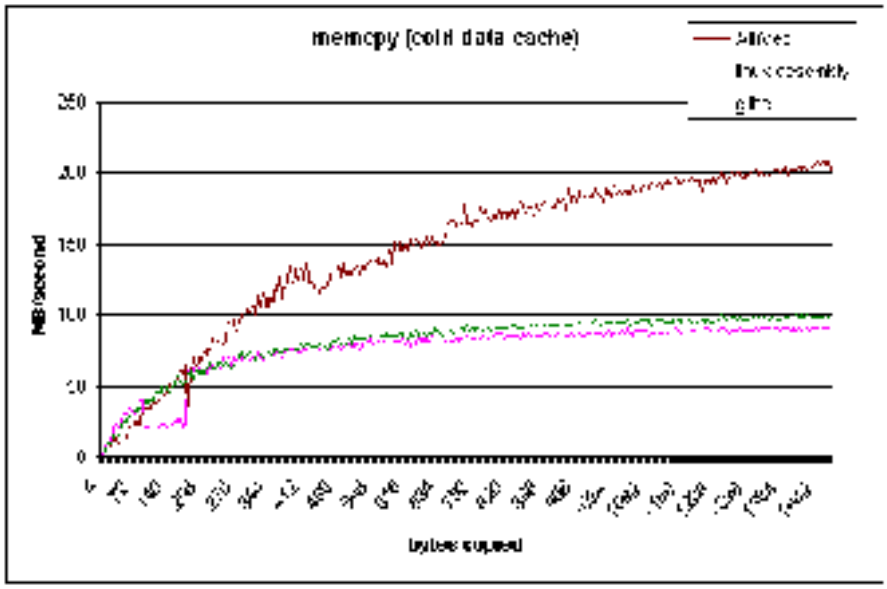

Figure 5. memcpy Throughout with Cold Data Cache

Figure 6 and Figure 7 show the performance of the checksum and memcpy() functions with both warm and cold data caches. Again, AltiVec-enabled functions provide significant speed up for larger data sizes. It is important to notice that the checksum computation is nearly free for AltiVec if combined with memcpy. The reason is that, in the memcpy loop, all of the data is already resident in the quad-word AltiVec vector registers. Therefore, computing checksum with the resident data only takes three single-cycle instructions

to execute. On the contrary, the scalar version memcpy is 10 to 20 percent slower if combined the with checksum.

As shown in Figure 6, the combined memcpy and checksum routine coded with AltiVec instructions is about four times faster than the encoded scalar assembly routine. Notice that the performance and the actual execution sequence of the combined checksum and memcpy C code is very compiler-dependent. In Figure 6 and Figure 7, the C implementation combines two separate function calls instead of computing checksum in the same loop as memcpy.

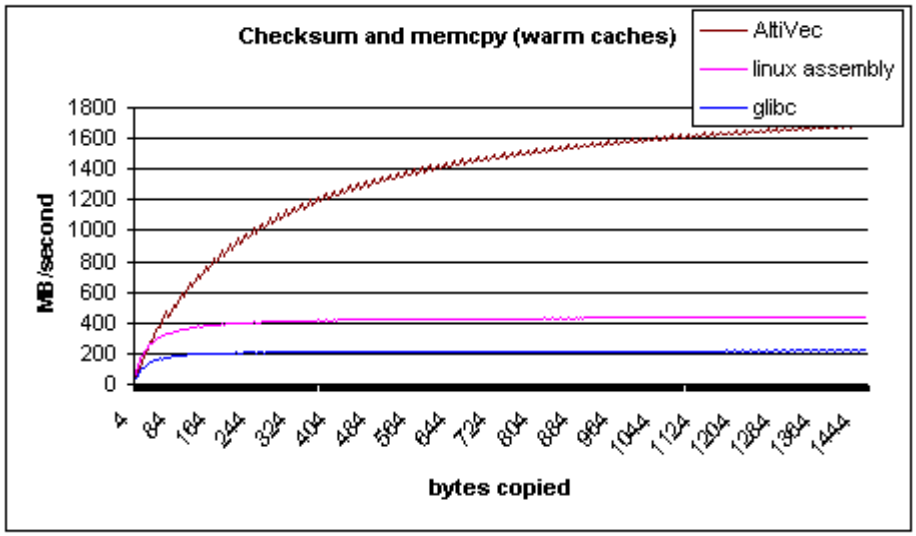

Figure 6. checksum_and_memcpy Throughput with Warm Data Cache

The data stream touch **(dst)** instruction is used in checksum_and_memcpy(), the result in Figure 7 shows that a speed up of more than 2 times can be achieved with medium sized data packets.

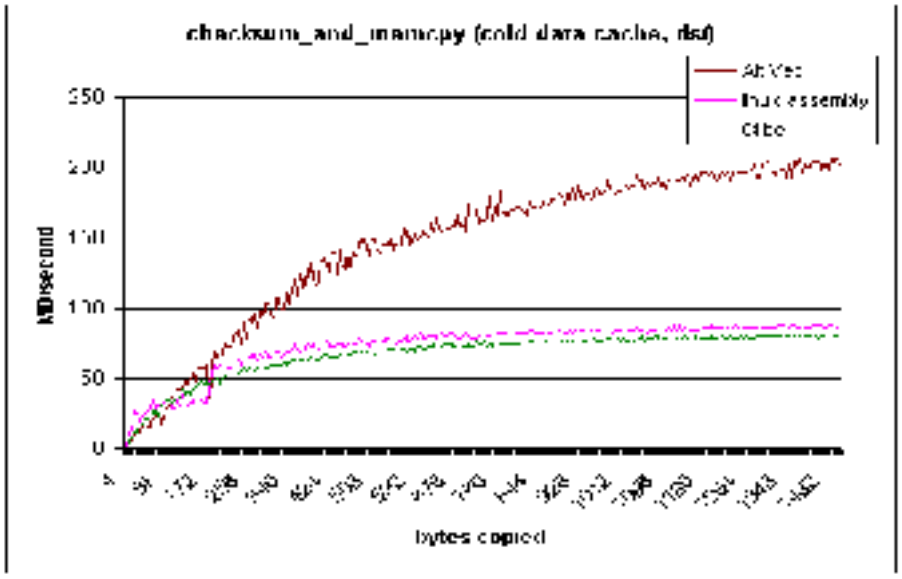

Figure 7. checksum_and_memcpy Throughput with Cold Data Cache

Benchmark Results

To show how these AltiVec-enabled library functions can benefit TCP performance in the real networking environment, a TCP benchmark was used to evaluate the performance impact with the above three functions. Internal to the TCP benchmark, to simulate client-sever conversation, two tasks were created with their own TCP protocol control block (TCB), respectively. In a single-thread environment, the two tasks simply alternate in a busy loop.

Two simple emulation layers were created to measure the performance related to the TCP-network layer and the TCP-application layer interfaces. To avoid overhead associated with maintaining emulation layers, the implementation adopted the simple and effective mechanism described in the following paragraphs.

At the application emulation layer, all buffer descriptors are initialized as a ring buffer; thus, data can be quickly retrieved by the server/client task at run time. The network interface emulation layer is implemented as two double-linked lists with queue heads and tails stored in static data structures attached to the corresponding TCB. In addition to reducing overhead, this approach allows simulating network error conditions by simply changing status bits in buffer descriptors or by swapping pointers for out-of-order conditions.

Figure 8 shows the architecture of the TCP benchmark. Each TCB maintains an event-driven state machine for connection management and exception handling. Data flow in the egress direction includes two queues: the unsent queue and the unacknowledged queue. Input data from the network emulation layer is checked for matching address information and data integrity. No further data processing is performed.

In summary, this TCP benchmark models all normal data transactions, segmentation, queuing, and connection management. Because network anomalies, such as corrupted, duplicate, lost, and out-of-order packets, occur rarely [2] in the real network, modeling of those exception cases is not included in this TCP benchmark.

The following five standard tests were available in the benchmark to simulate different network traffic patterns:

- The bulk data transfer test uses maximum standard TCP segment size (1460 Bytes) to emulate FTP like data flow.
- The interactive data transfer test uses small data packets to simulate TELNET or RLOGIN type of traffic
- The mixed packet size test is based on statistics collected in real network traffic
- The connection request/response test is configured such that the connection setup and close process is repeatedly performed in the loop with no data exchanged.
- The jumbo frame size test uses a non-standard gigabit Ethernet jumbo frame of 8000 bytes.

Figure 8 shows how the network interface data structure (NIF) models the data as necessary for interacting with the network and lower layers. Some data passing arrows in the graph include computing a partial checksum in memcpy.

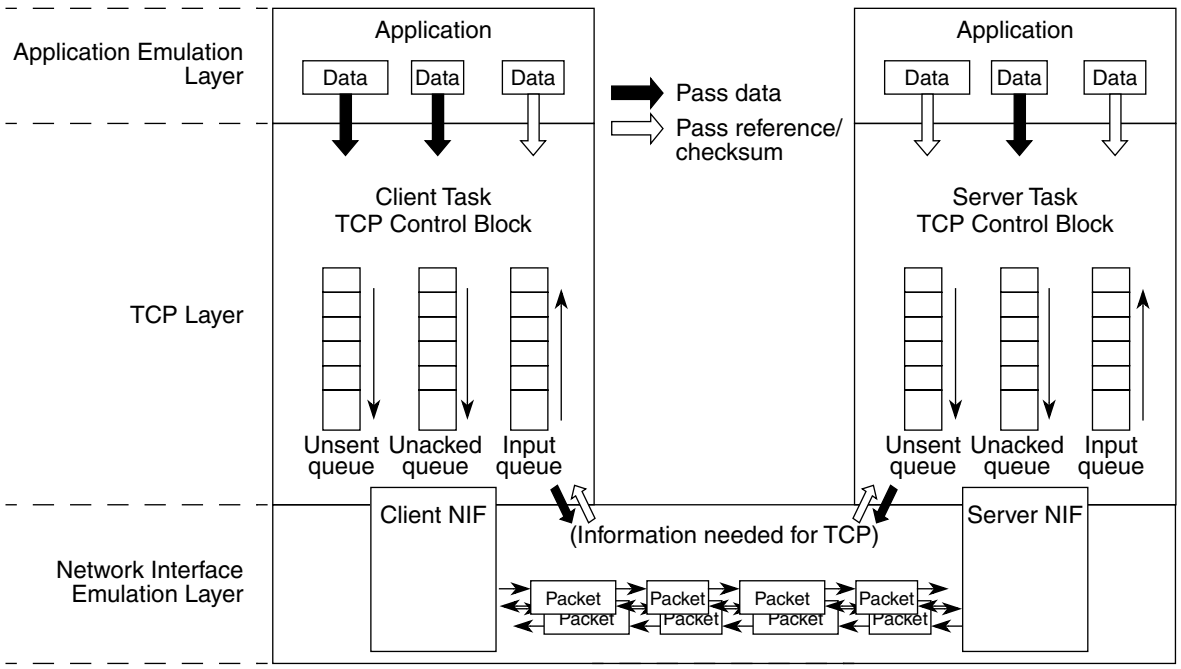

Figure 8. TCP Benchmark Internal Architecture

The profile result of the TCP benchmark is shown in Figure 9. It shows the percentage of execution time consumed by C checksum and memcpy routines [1] on a Solaris system [2]. Data is generated by a GNU profiler 2.12.1. Due to the multitasking nature of the Solaris platform, this result may vary slightly in later test cases (on a single-task environment).

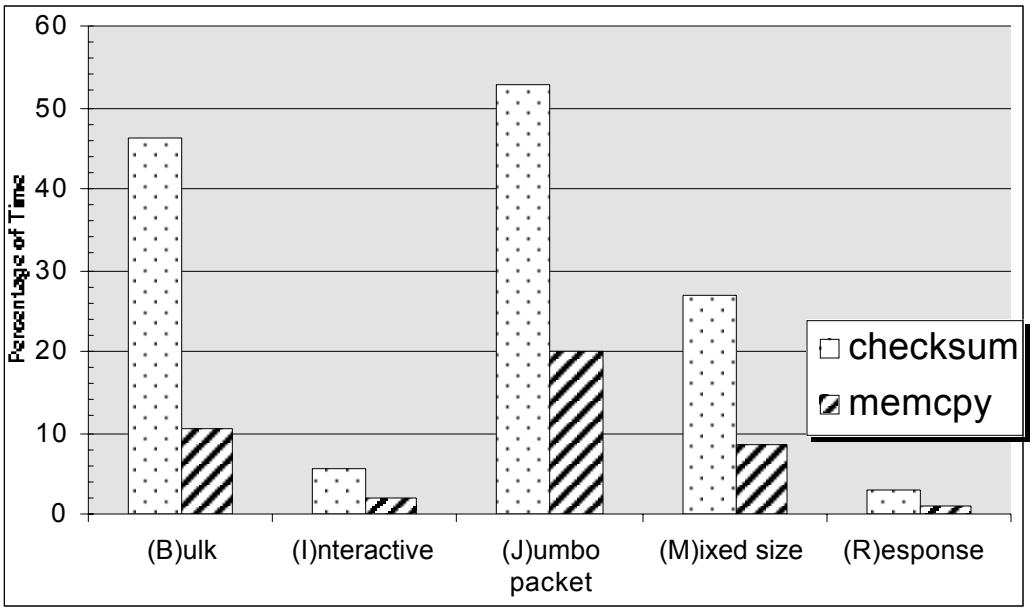

Figure 9. Profile Result of TCP Benchmark on Solaris

[1] C and GLIBC compiled by Solaris native GNU C compiler version 2.95.3
[2] Sun OS 5.8

As shown in Figure 11, in three of the five standard test cases, AltiVec-enabled library functions achieved significant speed-up over scalar C and assembly code.

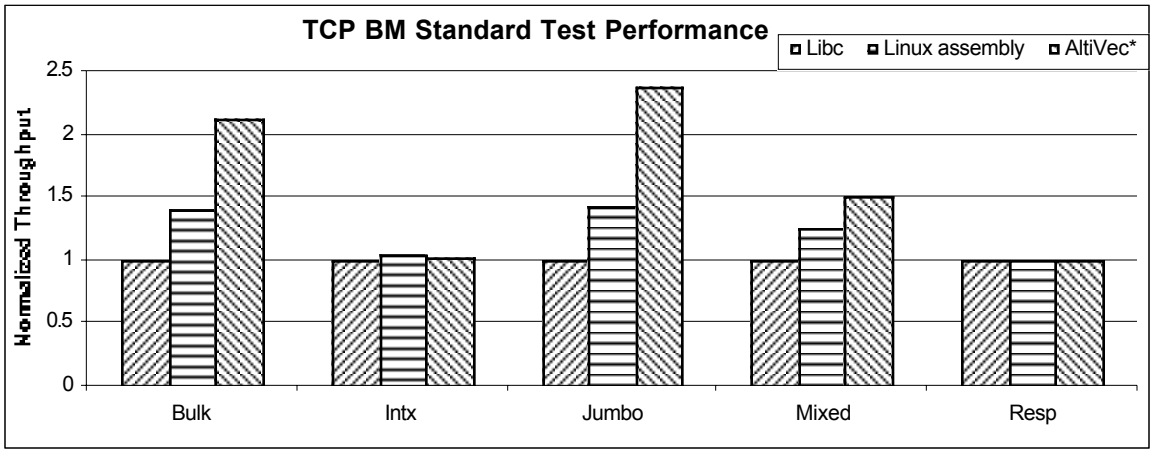

Figure 10. TCP Benchmark Standard Test Performance (Normalized)

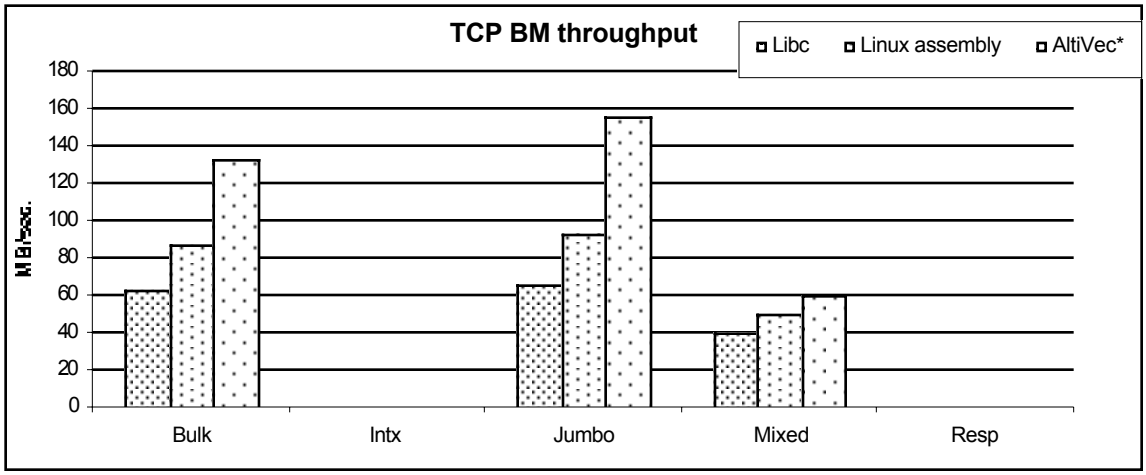

Figure 11. TCP Benchmark Standard Test Result (Normalized)

Because interactive data flow and response time tests consist mainly of signaling packets and tiny data packets (averaging 1 to 2 bytes of payload in the interactive data flow), little performance difference can be observed in those two cases. Therefore the throughput is minimal as compared to the other three cases. This characteristic is easily seen in Figure 11, which shows the actual data throughput for the five tests. In reality, the round trip time is more important in the interactive data flow and response time tests, whereas data throughput is the focus of the bulk data flow cases with either a standard packet size, a jumbo frame size, or in a mixed packet environment.

Concluding Remarks

This paper presents the benefits of applying AltiVec technology in today's challenging network computing environment. A systematic and quantitative approach is illustrated with the enhancing TCP performance example. In summary, AltiVec can be used whenever multiple streams of data can be operated on in parallel or when large data blocks need to be processed.

An AltiVec implementation of a function may vary from its scalar version, but a software abstraction layer can be used to isolate the code changes. Ultimately, AltiVec-enabled library functions can be linked in with minimal porting effort.

Another key to optimization is to detect the opportunity to properly vectorize generic scalar code. An in-depth knowledge of the target environment and efficient profiling tools can help detect the bottlenecks. Along with knowledge of the AltiVec architecture, potential performance enhancement can be achieved.

References

[1] Ramon Cacerest, Peter B. Danzig, Sugih Jamin, Danny J. Mitzel. Characteristics of Wide-Area TCP/IP Conversations. CSD, Univ. of Southern California, 1991

[2] Jeffrey C. Mongul. Observing TCP Dynamics in Real Networks, WRL, DEC, 1992

[3] V. Paxson. Known TCP Implementation Problems, RFC2525. Network Working Group, 1999

[4] W. Richard Stevens, TCP/IP Illustrated, 1994

[5] Jon B. Postel. Transmission Control Protocol, RFC793, Network Information Center, SRI International, 1981

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140
(800) 441-2447

JAPAN:

Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu Minato-ku
Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre, 2 Dai King Street
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong
852-26668334

TECHNICAL INFORMATION CENTER:

(800) 521-6274

HOME PAGE:

http://www.motorola.com/semiconductors

AltiVecTCPIPWP/D

Freescale Semiconductor, Inc.