Technical Articles

Using AltiVec™ Technology to Accelerate Network Software in a Linux Environment

Author: Jacob Pan  
Topic: AltiVec technology  
Date: Aug 22, 2004  
Revision: 1.0

In most protocol stack implementations, the bottlenecks in large throughput traffic are data movement and checksum computation. AltiVec™ technology can dramatically accelerate data movement/processing, which in turn can reduce the bottlenecks. Faster data movement can further reduce buffer polling time, memory allocation/deallocation delays, and queuing time. Ultimately, a better orchestrated and efficient buffer management system can be realized for networking protocols.

In order to take advantage of AltiVec technology for a specific algorithm, many software engineers develop AltiVec code in either intrinsic C or assembly language. Another approach is to use the open source AltiVec library, which contains many process-intensive C library functions, as well as some commonly used algorithms such as those that compute an IP-style checksum. The latter approach requires much less coding effort while still preserving the portability of existing software.

Currently, AltiVec is enabled in most commercial Real-Time Operating Systems (RTOSs). However, the scope and AltiVec support mechanisms for different environments, such as Linux, may vary greatly, which in turn may affect the performance of different applications. This article provides details about how to leverage AltiVec technology to improve the performance of networking applications that run on Linux.

AltiVec technology and Linux 2.4

The Linux kernel supports AltiVec technology in the user mode only. The kernel space machine state register (MSR) setting prohibits AltiVec usage; the user space MSR disables AltiVec by default. When a user process executes an AltiVec instruction, it first takes an 'AltiVec unavailable' exception, which leads program execution to enter kernel mode. The corresponding kernel handler then enables AltiVec for that process when it returns to user mode. After that, a lazy context switch is performed between AltiVec-enabled processes. No more AltiVec exceptions should occur for the process that executed the AltiVec instruction. This approach is nearly identical to the way floating point registers are handled.
The following C code illustrates the control logic in the kernel process switch function:

```c
if ((previous_process->thread.regs
    &&(previous_process->thread.regs->msr& MSR_VEC)))
    giveup_altivec(previous_process);
```

The current AltiVec registers are saved in the `giveup_altivec()` function. A simplified scenario is illustrated in Figure 1.

![Figure 1: Linux AltiVec Support](image)

Managing AltiVec context independently

Managing AltiVec context independently at the function level allows the use of AltiVec functions without operating system dependency. This approach provides AltiVec users with another option that may be suitable for their specific applications.

The new approach is implemented in assembly macros, which can be inserted in AltiVec assembly functions as prolog and epilog. C functions can also declare the assembly macros as inline. This implementation includes the following components:

- A counting semaphore-like atomic counter that measures the number of AltiVec function calls at the time of execution
- A global independent stack with maximum size = 4
- Stack operation primitives with a binary semaphore lock for stack operations
To use the assembly macros, Altivec software developers need to provide two general purpose registers (GPR) for the macros’ performance atomic memory access and stack operations. A convenient choice is a pair of reusable volatile registers. With the epilog and prolog appended, the overhead of kernel switching is now transferred to each function, but since nested Altivec function calls occur rarely in most applications, the overhead for each function call is just a pair of atomic load and stores that most likely access a cache-resident global counter.

When testing inside an OS kernel, the independent Altivec stack can be initialized during booting. The stack can be allocated with kmalloc() function to ensure a contiguous physical memory space. The stack is then cleared, and the counter is set to zero. Alternatively, stack space can also be statically allocated. Stack size need to be adjusted to avoid possible overflow. In our test, size limit of four is sufficient. The code can be written so that the scalar version of the function is used when a stack overflow occurs, or the last Altivec function to access the stack can be put into busy spin until more stack space becomes available.

The source code format is shown in the following pseudo code:

```
Altivec_function_begin: // prolog
INCREASE_VEC_USAGE_COUNTER(r6, r7);

Function_body:
    ......

Altivec_function_exit: // epilog
DECREASE_VEC_USAGE_COUNTER(r6, r7);
```

If an Altivec function has multiple exit points, the same epilog can be inserted before all non-conditional exit points. All conditional exit points have to be merged into a single point where the epilog can be inserted. Figure X illustrates the control flow of the new approach.

Since Linux enforce a kernel and user boundary, simply using Altivec functions with this approach inside the kernel will exclude the usage of Altivec in the user land. Additional modifications are needed to the kernel to make sure coherency of Altivec context among user and kernel processes.
Implementing AltiVec functions

Abstracting code into layers is important for software portability; it allows for the reuse of common code for multiple projects. Portability increases as hardware dependency decreases. On the other hand, the complexity increases and performance decreases as more software layers are created. Often times in order to use new hardware technology, hardware-dependent code must be developed. If a performance bottleneck can be identified in a few isolated function calls or code fractions, those functions can often be substituted with optimized code.
Therefore, performance enhancement can be achieved without extensive rewriting of the existing software. Networking software typically includes a socket layer that is defined in the operating system application programming interface (API). The majority of APIs are somewhat compatible with the POSIX standard. The socket layer then provides a common interface between the application and the protocol stack. Optimizing data movement in this common layer can benefit all communication protocols.

Linux is an open source operating system, and its protocol stack is derived from the commonly used Berkeley Software Distribution (BSD) UNIX operating system. We will examine how to port AltiVec functions to Linux networking code and then test those functions. Again, AltiVec-enabled IP checksum and standard memcpy routines are used in code examples. The first step is to insert the AltiVec assembly source functions into the following appropriate files in the kernel source tree:

```
linux_source_top_dir/arch/ppc/lib/string.S
linux_source_top_dir/arch/ppc/lib/checksum.S
```

The original scalar version library functions should be preserved because completely replacing them with AltiVec functions may result in AltiVec lowering performance or creating problems. For example, if memcpy is replaced with the AltiVec version, it may cause one or more of the following problems:

- Too many processes have to support AltiVec context, causing excessive overhead.
- The AltiVec unit is used during boot time before caches are fully configured.
- AltiVec I/O or non-cacheable memory is used, resulting in either a slowdown or an alignment problem.

Therefore, naming the AltiVec version with a suffix such as “_vec” allows selective usage of AltiVec routines for optimal performance and controllable behavior. For example, the following two functions are inserted into string.S:

```
GLOBAL(memcpy_vec)
GLOBAL(__copy_to/from_user_vec 1)
```

Unlike memcpy, the IP checksum function is used only in the networking software (mainly in the protocol stack and its buffer management.) This function can be completely replaced with the AltiVec version. Although all Freescale G4 PowerPC processors support snooping on the front side bus, attention must be paid to ensure that the correct mapping of network buffers in cacheable memory is provided. As a result, the following two functions in checksum.S are vectorized:

```
csum_partial_copy_generic(src, dst, len, sum, src_err, dst_err)
csum_partial(buff, len, sum)
```

Hardware abstraction layer access functions also have to be modified to accommodate the changes. The routines can be copied from:

```
linux_source_top_dir/include/asm-ppc/uaccess.h
```
Networking code changes are fairly simple since all the checksum routines have been replaced with the AltiVec versions, and the impact is transparent. On the other hand, memcpy_vec() is selectively used in the socket buffer management code. Two socket buffer management files are listed below:

```
linux_source_top_dir/net/core/iovec.c
linux_source_top_dir/net/core/skbuff.c
```

The Linux kernel can be rebuilt after the above changes have been made. There is no need to change the make system or configuration files.

**The results are in**

Benchmarking was done by Freescale to determine the benefits of leveraging AltiVec in a network application. Many factors can affect network throughput, delay, and packet rate. Since throughput-critical traffic is almost always carried by large packets, the focus of this experiment is on TCP/UDP bulk data transfer over the standard Ethernet.

Network throughput data is gathered in conjunction with the host processor workload. A Linux kernel with an AltiVec-enhanced socket and protocol stack is compared with the original scalar version on identical hardware. Throughout the test, Netperf is used to benchmark the overall network throughput of the TCP/UDP traffic. PowerPC performance monitoring counters and other software tools are also used for CPU efficiency analysis. Results show that AltiVec-enabled data processing functions contribute to better performance in both throughput and CPU efficiency.

As shown in Figure X, the test bench consists of two identical Freescale Sandpoint evaluation systems. Each system is equipped with identical PowerPC processors (both MPC74XX and MPC745X processors were tested) but loaded with separate Linux kernels with and without AltiVec enhancement. Netperf-generated TCP/UDP traffic is tested across the Gigabit Ethernet channel through a PCI-based network interface card.

To explore the performance enhancement caused by AltiVec, test cases were divided into the following three dimensions:

- Different protocols, namely TCP and UDP
- Different socket buffer sizes
- Client versus server

![Figure 3: Benchmark System Configuration](image-url)
Socket buffer size plays an important role in achieving better TCP performance; it is often a trade-off between memory consumption and performance. However, studies have shown that the buffer size should be no smaller than the bandwidth and delay product. In addition, its relation with cache sizes may also affect cache hit ratio and consequently affect overall performance.

Figure 4 shows CPU usage variation with respect to different socket buffer sizes. The default socket buffer size is 16 KB, which is designed for fast Ethernet throughput. With little variation in throughput of around 328 Mbps, Figure 6 shows that CPU utilization varies greatly as buffer size changes. The AltiVec-enhanced kernel shows the best performance enhancement with a socket size of 128 KB, which is the closest to the bandwidth delay product for Gigabit Ethernet. At socket buffer size 32 KB, which is the same as the L1 cache size, performance degraded for both cases with and without AltiVec technology.

![Figure 4: Netperf TCP Stream Test with Different Socket Buffer Sizes](image)

The reason why there is little difference in the throughput is that TCP is a connection-oriented protocol; its throughput is limited by the performance of both connection endpoints as well as network delays. On the other hand, the UDP test results shows a throughput increase from using AltiVec functions. Since UDP is a connection-less and best-effort protocol, the Netperf benchmark continues sending data without a payload checksum and does not wait for acknowledgement. Therefore, performance does not vary with respect to different buffer sizes. At runtime, the AltiVec optimization point is limited to the memcpy() function only. No payload checksum is involved.

Another important factor that may affect AltiVec enhancement is the internal segmentation and buffer management mechanism. When processing bulk data transfer, data should be copied into a buffer with sufficient head room to grow and also utilize the maximum segment size that the physical path allows. For example, Linux segments the TCP bulk data stream into 1460 byte long data packets, which is the maximum data size allowed by the path maximum transfer unit (MTU). In this case, the AltiVec-enabled checksum and memcpy() functions can be utilized to process data movement for excellent performance.
User data is segmented into various sized data buffers or clusters. When data movement occurs between software layers, AltiVec functions can perform better in large data blocks. It is important to avoid unnecessary fragmentation of the data. As shown in Figure 10, even though two packets are carrying the same amount of data, performance is impacted differently. With AltiVec, data movement is faster in packet 2 than in packet 1. This is because of the way data is fragmented. Packet 2 is represented in two large data clusters whereas packet 1 is represented in four smaller clusters. Allocating large data clusters wherever possible with room to grow for the potential packet headers can help reduce the number of small data fragments.

PowerPC processors provide hardware counters that can be used to measure important runtime statistics compiled on what happened inside the device. The measurement takes place without software intervention except when starting and stopping the counters. The following graphs show the performance monitoring data collected while running the Netperf TCP stream test. On a live TCP traffic test, the measurement is taken at the sender’s side.

The significant difference in measurement lies in the percentage of load and store instructions completed during execution time. Since AltiVec loads and stores are four times wider than its scalar version, the total loads and stores dropped from 49% to 38% in that particular test case, as shown in Figure 5. Because loads and stores are high-latency instructions compared to other arithmetic or branch instructions, they consume more time. Therefore, AltiVec enhancement can speed up overall data movement and enable faster buffer management, which results in better performance. Figure 6 shows the performance monitoring result of a TCP loopback test. In this test, Netperf and Netserver are run on the same host system, consuming nearly 100% CPU. Again, the AltiVec-enhanced version reduces loads and stores from 69% to 33%.

![Figure 5: Netperf TCP Stream (Live Traffic) Test Performance Monitoring Counts](image)

![Figure 6: Netperf TCP Stream (Loopback) Test Performance Monitoring Counts](image)
Performance monitoring tools provide direct insight into the processor, helping to explain why and from where the performance speedup is derived. Using the counters can also be helpful in debugging performance issues such as cache hit/miss ratios, which are closely related to AltiVec performance.

In summary, AltiVec can provide major performance enhancement to network applications in a Linux environment where large data blocks or multiple data streams can be processed in parallel. Its speedup can be extended beyond the networking example described in this document to any application with similar parameters.

Resources

- Example AltiVec applications provided at www.simdtech.org
- *AltiVec™ Performance Enhancement in a Multiprocessing Environment* – the application note on which this article was based
- *Enhanced TCP/IP Performance with AltiVec* – white paper
- More AltiVec technology information at:
  - www.freescale.com/altivec
  - Apple Computer’s Developer Site – Velocity Engine

Author information

Jacob Pan is a PowerPC applications engineer for the Computing Platforms Division of Freescale Semiconductor.