**Freescale Semiconductor**
Application Note

Document Number: AN2581
Rev. 1, 07/2006

# Altivec™ Performance Enhancement in a Multiprocessing Environment

*By Jacob Pan*
*Networking and Computing Systems Group*
*Freescale Semiconductor, Inc.*

This application note addresses the process and methodology of porting AltiVec™-enabled software to a multiprocessing operating environment. The focus of this application note is not only on speeding up performance in individual applications, but also on the techniques for enabling AltiVec in operating systems. The Linux and VxWorks kernels, combined with their networking stacks, are used as examples to illustrate the porting process. Various operating system support options are discussed, along with an analysis of their performance issues.

To put the AltiVec software porting methodology into practice, a real network traffic test was performed on a Freescale G4 PowerPC™ with AltiVec-enabled Linux and a VxWorks protocol stack. Performance increases were measured in CPU efficiency and absolute throughput.

**Contents**





freescale™ semiconductor

# 1 Altivec Basics

AltiVec technology delivers a performance boost for many process-intensive applications, such as high-throughput TCP/IP networking[1]. Software techniques for enabling AltiVec can be as simple as linking with AltiVec library functions or as significant as recoding entire algorithms with AltiVec assembly instructions. Performance speedup is often tied to the software porting effort. The challenge is using AltiVec efficiently in multitasking environments.

Most embedded systems run some kind of operating system that supports multiprocessing, which permits concurrent execution of multiple processes on the same hardware. Between time slices and interrupts, context switching is essential for sharing hardware resources and maintaining coherency among running processes. Although the exact content of a process varies among different operating systems, a typical process contains the following:

- A thread of execution, that is, a process program counter
- The CPU register file and optional floating-point registers
- A stack for dynamic variables and function calls
- I/O assignments for standard input, output, and error
- Timers for delays and time-slices (optional for real-time operating systems (RTOS))
- The kernel control structures
- Signal handlers
- Independent memory space (optional)
- Debugging and performance monitoring values (optional)

The CPU register file is a central part of the hardware context. Because context switching may occur frequently, effectively saving and restoring registers is essential to performance. The introduction of the AltiVec unit and its own vector register file expands the hardware context by up to 32 quad words. In addition, AltiVec registers must be protected if used in multiple processes concurrently. Stack operations are also needed for nested calls to AltiVec-enabled functions. The AltiVec unit introduces some overhead, which can vary greatly at runtime depending on the context management mechanism, the system clock tick frequency, and interrupts.

The performance issues of different techniques used by operating systems are analyzed in Section 3. An AltiVec application programmer should be acquainted with these techniques and use them properly to achieve highest performance.

# 2 Enabling AltiVec in a Multiprocessing Environment

Although AltiVec is enabled in most commercial RTOSs, the scope and AltiVec support mechanisms of different environments may vary greatly, which in turn affects the performance of different applications. For example, the Linux and VxWorks operating systems represent two distinct methodologies that have performance trade-offs. Before executing code on the AltiVec unit, you must understand how AltiVec is supported in these multiprocessing environments.

1. See TCP/IP white paper, “Enhanced TCP/IP Performance with AltiVec,” 2003.

## 2.1 Linux 2.4

The Linux kernel supports AltiVec 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:

```
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.

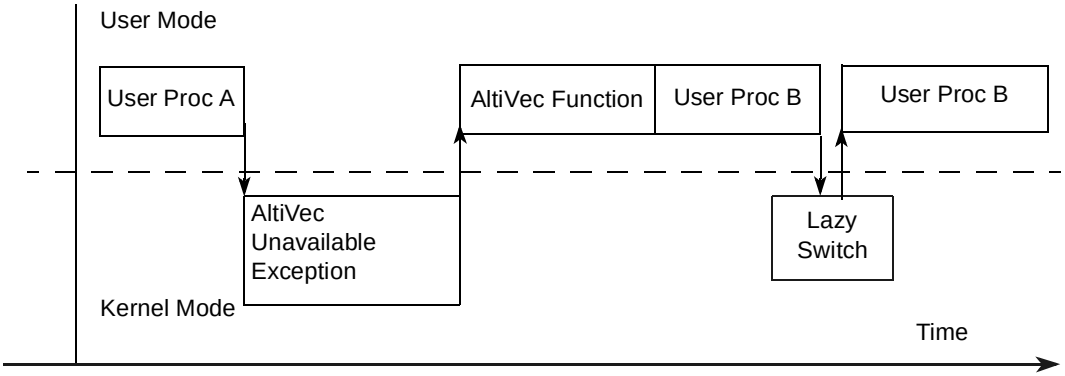



**Figure 1. Linux AltiVec Support**

The benefit of this approach is that the kernel can automatically handle AltiVec context as soon as the user program needs it. It is not necessary for the user program to tell the kernel which process is using the AltiVec unit. From a performance perspective, it is inefficient for the AltiVec unit to be repeatedly used in newly spawned processes; performance may be offset by excessive AltiVec exceptions. Additionally, this approach excludes AltiVec from some important areas, such as networking stacks because most of the core networking stack is built into the kernel. Because Linux by default does not support AltiVec usage in kernel mode, such areas cannot benefit from AltiVec unless AltiVec content is managed independently. Section 2.3, "Independent Approach for Managing AltiVec Context" introduces such an approach.

## 2.2 VxWorks

For better performance and to meet real-time constraints, some RTOSs do not force kernel and user boundaries. For example, in VxWorks®, all tasks execute in privileged mode, and memory spaces are shared among tasks. The AltiVec support method in VxWorks 5.5 does not use the AltiVec unavailable exception to trap user programs. It relies on the user program to notify the kernel explicitly whether the current task is using the AltiVec unit. The VX_ALTIVEC_TASK flag must be set in the options field[2]

2. VxWorks® for PowerPC Architecture Supplement 5.5.

whenever a new task is spawned with AltiVec function calls. Otherwise, the system takes an AltiVec unavailable exception, and the task is suspended. VxWorks run-time support for AltiVec enables the usage of a vector unit in both the kernel and user programs. Networking and other kernel functions can take advantage of AltiVec as long as the vector unit is initialized and the running task is spawned with the AltiVec option.

Because VxWorks uses a slower system tick[3] in a preemptable kernel, context switching is minimal if AltiVec usage is limited to a single task with few interrupts. However, there can be significant overhead under some scenarios such as networking applications in which the kernel receives frequent interrupts. Because knowledge of AltiVec usage occurs only at the task level, the kernel often performs unnecessary save/restores on the AltiVec registers even when the current task is not executing AltiVec functions.

Consider the scenario in Figure 2, which consists of two running tasks both spawned by VX_ALTIVEC_TASK. Both tasks include some AltiVec code (represented by the shaded areas). AltiVec function fractions occupy portions of execution time, and it is rare for context switching and interrupts to occur in the middle of function execution. Therefore, since the kernel has no knowledge of the beginning or end of AltiVec usage, it has to save and restore VRs every time context switching is needed. In most cases, this is unnecessary and hinders performance and interrupt latency.

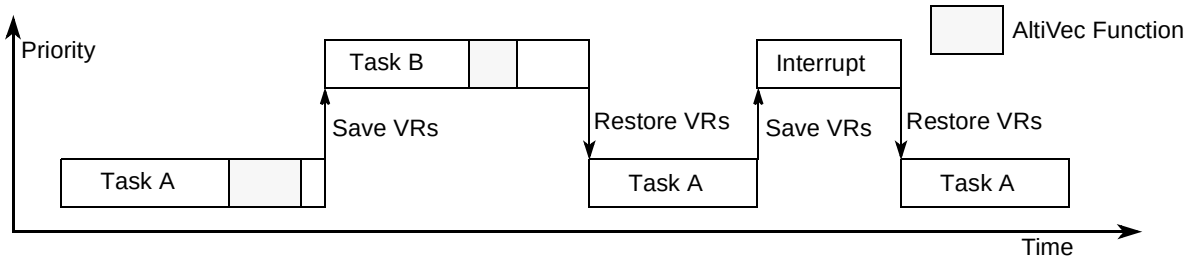



**Figure 2. AltiVec Context Switching in VxWorks**

## 2.3 Independent Approach for Managing AltiVec Context

A new approach that manages AltiVec context independently at the function level is aimed at solving the usage and performance limitations discovered in the Linux and VxWorks kernels. This approach offers AltiVec users another option that may be suitable for their specific applications. Above all, it allows the use of AltiVec functions without operating system dependency. This 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

When testing in the VxWorks environment, the global independent AltiVec stack is initialized in the user application initialization function. The stack is cleared, and the counter is set to zero. Memory space is statically allocated for the stack and semaphores. Stack overflow is possible due to the stack size limit of four but is unlikely. The code can be written so that the scalar version of the function is used when a stack

3. 60 Hz by default.

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
        INCREMENT_VEC_USAGE_COUNTER(r6,r7);
        Function_body:
        ......
Altivec_function_exit: // epilog
        DECREMENT_VEC_USAGE_COUNTER(r6,r7);
```

If a 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 3 illustrates the control flow of the new approach.

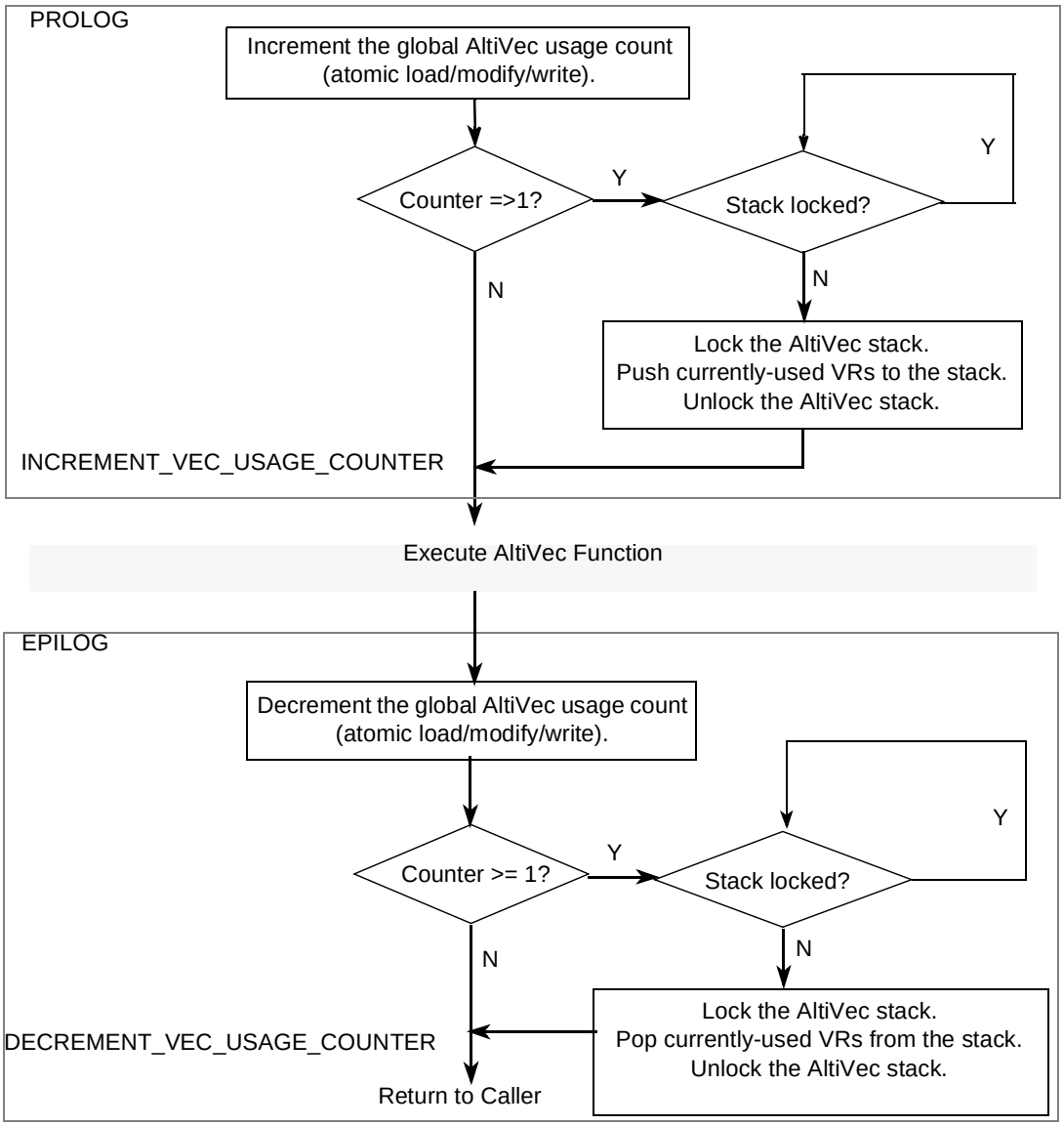



**Figure 3. Control Flow of Independent AltiVec Context Management**

To use the assembly macros, AltiVec software developers must provide two general-purpose registers (GPR) for the macro 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. Performance numbers of networking benchmarks will be provided in Section 3.3, where an AltiVec-enabled protocol stack is tested.

## 2.4 Summary of OS Support Options

Awareness of potential kernel overhead is important for optimizing performance with AltiVec-enabled software. System clock ticks and interrupts determine the frequency of saving and restoring AltiVec context. Non-preemptable kernels often use faster ticks to accommodate the delay derived from blocked processes with real-time requirements. On the other hand, preemptable kernels tend to use a slower tick that reduces the overhead of saving/restoring AltiVec context. Interrupts also require saving and restoring hardware context. They may become significant in some applications, such as networking traffic with a large amount of small packets. This challenge is not unique to AltiVec but is an issue general to operating systems; for the same reason, more and more network devices and drivers implement optimization schemes to cope with flooding interrupts. Usually, system interrupts can be reduced by using packet-gathering hardware or polling-based drivers.

Table 1 summarizes the performance trade-offs among the options for supporting AltiVec in a multitasking environment. Overhead comes mainly from saving and restoring vector registers. The column, “Lower overhead with less frequent interrupts,” refers to applications such as graphics editors, which have infrequent interrupts. The column, “Higher overhead with more frequent interrupts,” refers to such applications as the file transfer protocol (FTP), which require frequent interrupts. The column, “Low overhead with large data block to process,” refers to applications such as image processing software, where a large amount of data is processed but there are not many interrupts. As the table shows, running an application with frequent interrupts with the independent approach is more efficient than with the Linux-like and VxWorks-like schemes; vector registers need only to be saved with nested AltiVec function calls.

**Table 1. Comparison of AltiVec Multitasking Support**

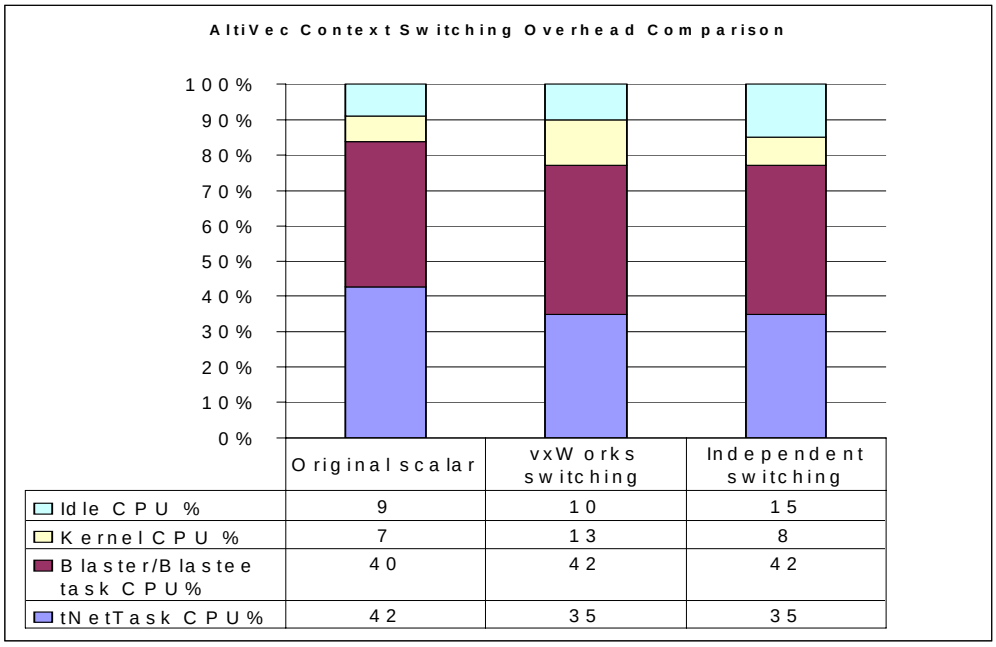
| Context Management Schemes | AltiVec Usage in Kernel | Kernel AltiVec Dependency | Overhead with respect to application types | | |
|---|---|---|---|---|---|
| | | | Lower Overhead with Less Frequent Interrupts | Higher Overhead with More Frequent Interrupts [1] | Low Overhead with Large Data Block to Process |
| Linux-like | No | Yes | Yes | Yes | Yes |
| VxWorks-like | Yes | Yes | Yes | Yes | Yes |
| Independent | Yes | No | Yes [2] | No | Yes [3] |

1 Only if running more than one AltiVec task concurrently. Low level interrupt service routines typically should not use AltiVec.

2 Normally low. High only when AltiVec functions are called frequently with small amounts of data to process, which is not recommended.

3 Same as 2

Figure 4 shows the differences in overhead among context switching schemes in VxWorks. A test was executed on a Freescale Sandpoint system with a MPC7455 processor. A networking benchmark was used to generate busy TCP/IP traffic over the Ethernet. In the AltiVec cases, some bcopy()[4] and all IP checksum routines were replaced with their AltiVec counterparts. All networking tasks were spawned with the AltiVec option in the VxWorks task level switching test, whereas no AltiVec tasks were spawned in the other two cases (scalar and independent). Due to frequent interrupts, the kernel CPU usage increased from 7% to 13% with task level switching. In contrast, the independent approach introduced negligible overhead to the networking tasks and nearly unchanged kernel overhead (%1).




| | Original scalar | vxWorks switching | Independent switching |
|---|---|---|---|
| Idle CPU % | 9 | 10 | 15 |
| Kernel CPU % | 7 | 13 | 8 |
| Blaster/Blastee task CPU% | 40 | 42 | 42 |
| tNetTask CPU% | 42 | 35 | 35 |

**Figure 4. Overhead Comparison of AltiVec Context Switching Schemes in VxWorks**

Again, the actual overhead of each application may vary depending on the factors mentioned in this discussion; software implementation, such as device drivers, may also alter the outcome. No single approach is consistently superior to another. If AltiVec is guaranteed not to be used in interrupts, saving/restoring its context is unnecessary. Therefore, kernel overhead can be minimized, and interrupt latency remains unchanged.

# 3 Optimizing Networking Software with AltiVec

This section explains the techniques for porting AltiVec functions to an existing TCP/IP protocol stack. 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 reduces the bottlenecks[5]. 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.

4. A memory copy function with reversed destination and source pointers than libc memcpy()
5. TCP/IP white paper

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.

## 3.1 Software Abstraction

Abstracting code into layers is important for software portability; it allows for the reuse of common code for multiple projects. Using the HTTP or FTP server as an example, Table 5 shows common networking software layers. It helps to examine which parts of the software should be vectorized for both good performance and portability.

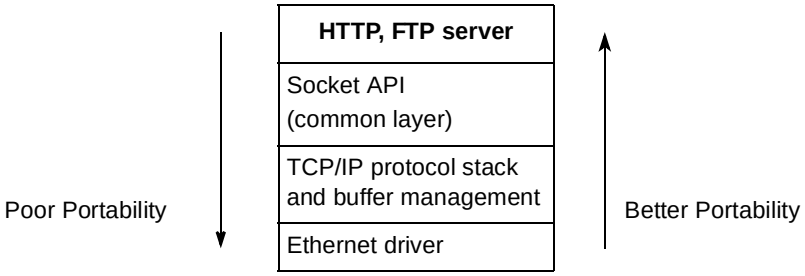



**Figure 5. Common Networking Software Layers**

Portability increases as hardware dependency decreases. On the other hand, the complexity increases and performance decreases as more software layers are created. Often, 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 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.

## 3.2 Linux Code Example

This section covers the details of porting to and testing AltiVec functions in Linux networking code. Linux is used for the code example because it is an open source operating system, and its protocol stack is derived from the commonly used Berkeley Software Distribution (BSD) Unix operating system. 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 lower performance or other 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_tofrom_user_vec[6])
```

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. Similar changes can be made to the VxWorks stack, which is also based on a BSD release.

## 3.3 Benchmarking/Testing on Network Traffic

Benchmarking network performance is a multi-dimensional project. 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 virtually 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

6. An exception table must be inserted for this function to deal with potential page faults caused by loads and stores.

for CPU efficiency analysis. Results show that AltiVec-enabled data processing functions contribute to better performance in both throughput and CPU efficiency.

### 3.3.1 Test Bench Configuration

As shown in Figure 6, the test bench consists of two identical Freescale Sandpoint evaluation systems. Each system is equipped with identical PowerPC G4 host processors[7] but loaded with separate Linux kernels with and without AltiVec enhancement. Netperf-generated TCP/UDP traffic is tested across the Gbit Ethernet channel through a PCI-based network interface card.

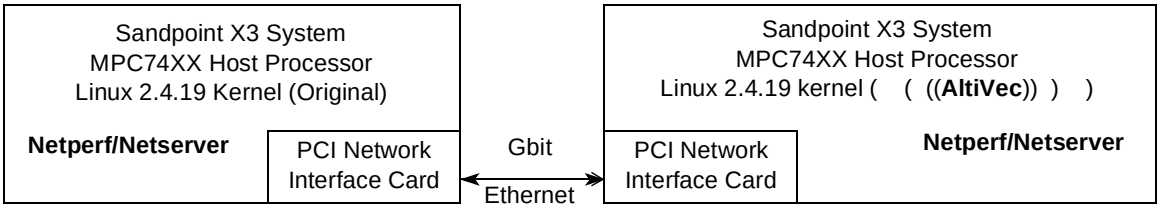



**Figure 6. Test Bench on Systems Loaded with Linux Kernels With/Without AltiVec**

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

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

### 3.3.2 Netperf Results

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

There is little difference in the throughput because TCP is a connection-oriented protocol; its throughput is limited by the performance of both connection endpoints as well as network delays.

7. Tested on both MPC74XX and MPC745X family processors on PPMC cards.

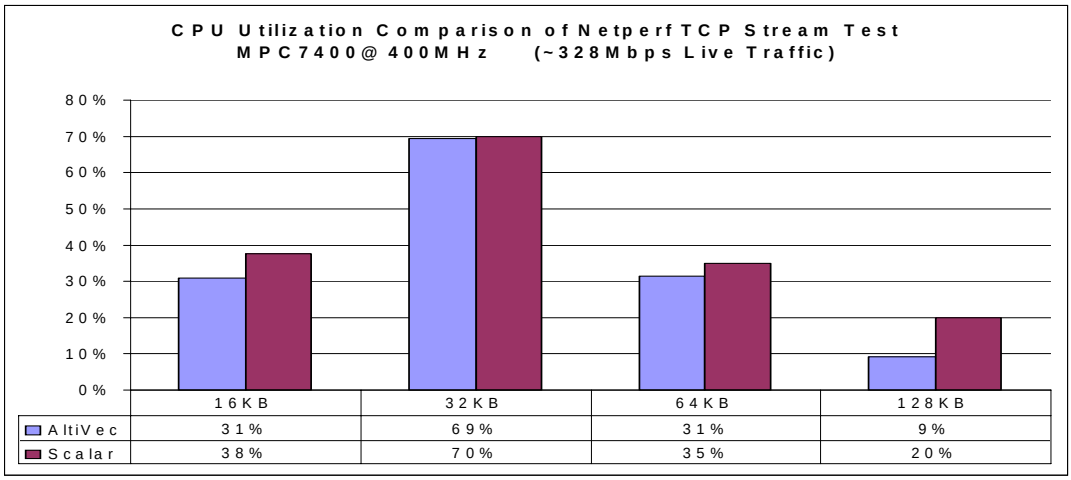



**Figure 7. Netperf TCP Stream Test with Different Socket Buffer Sizes**

On the other hand, the UDP test result in Figure 8 shows a throughput increase from using AltiVec functions. Because 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.

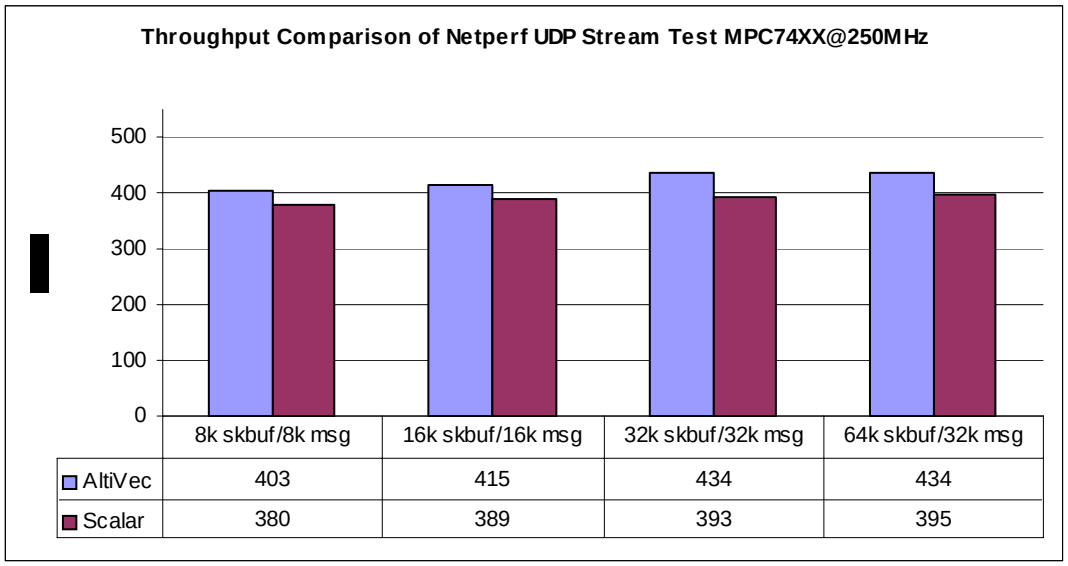



**Figure 8. Netperf UDP Stream Test with Different Socket Buffer Sizes**

Another important factor that may affect AltiVec enhancement is the internal segmentation and buffer management mechanism. When bulk data transfers are processed, data should be copied into a buffer with sufficient head room to grow and also use the maximum segment size that the physical path allows. For example, Linux segments the TCP bulk data stream into 1460 byte 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 used to process data movement for excellent performance.

A similar experiment was conducted in the VxWorks operating system. Collecting the data sizes used in AltiVec-enabled checksum and memcpy functions yields the following distribution, which is not the most favorable to AltiVec enhancement. The graph in Figure 9 was collected while running the Netperf TCP stream test (bulk data transfer.) It shows that most data segments processed by AltiVec functions are relatively small even though the final data packets are 1460 bytes long. Extra data fragmentation may have occurred due to the internal memory buffer cluster management scheme.

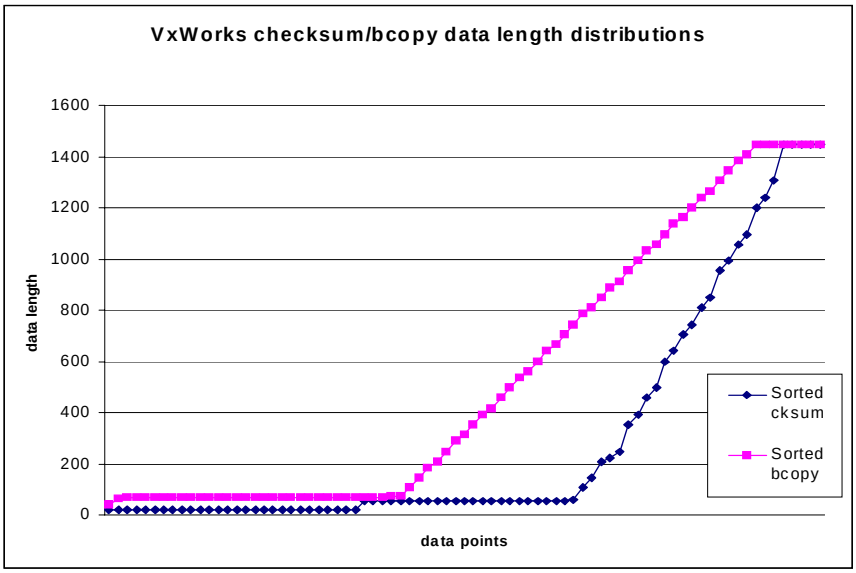



**Figure 9. VxWorks Data Segment Length Distributions**

User data is segmented into data buffers or clusters of various sizes. When data moves 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 affected differently. With AltiVec, data movement is faster in packet 2 than in packet 1 because of the way data is fragmented. Packet 2 is represented in two large data clusters but 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.

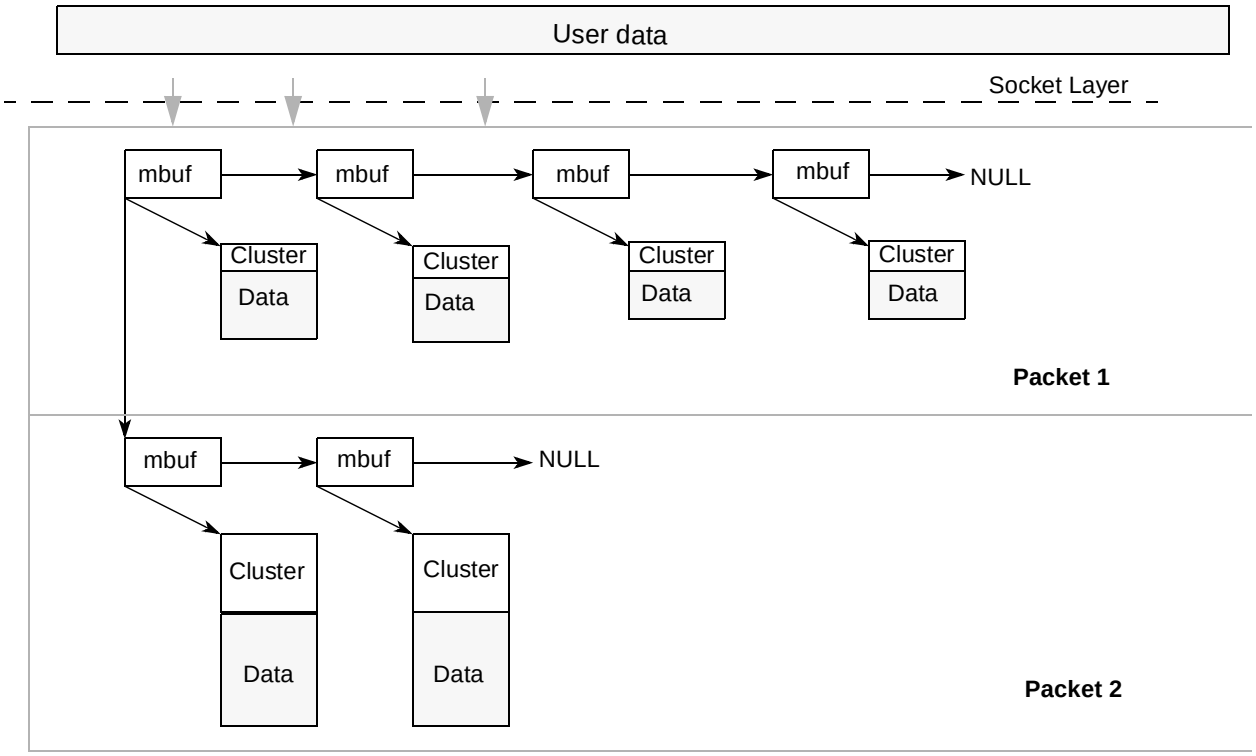



**Figure 10. Two Same-Size Packets with Different Memory Buffer Segmentation**

### 3.3.3 Analysis with Performance Monitoring Counters

PowerPC G4 processors provide hardware counters to measure important runtime statistics on 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 side; the significant difference lies in the percentage of load and store instructions completed during execution time. Because AltiVec loads and stores are four times wider than its scalar version, the total loads and stores dropped from 49 percent to 38 percent in that particular test case, as shown in Figure 11. 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.

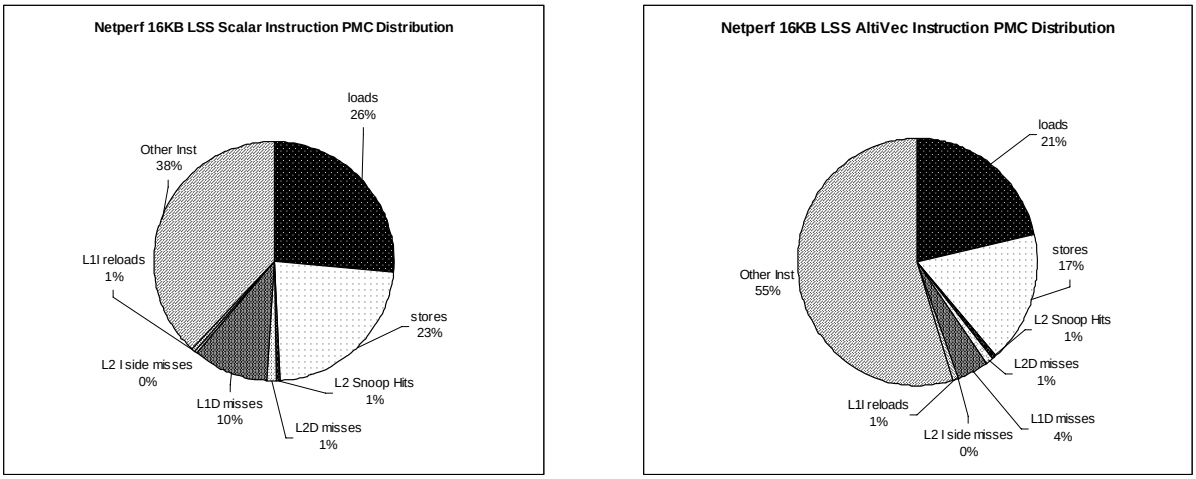



**Figure 11. Netperf TCP Stream (Live Traffic) Test Performance Monitoring Counts**

Figure 12 shows the performance monitoring result of a TCP loopback test[8]. Again, the AltiVec-enhanced version reduces loads and stores from 69 percent to 33 percent.

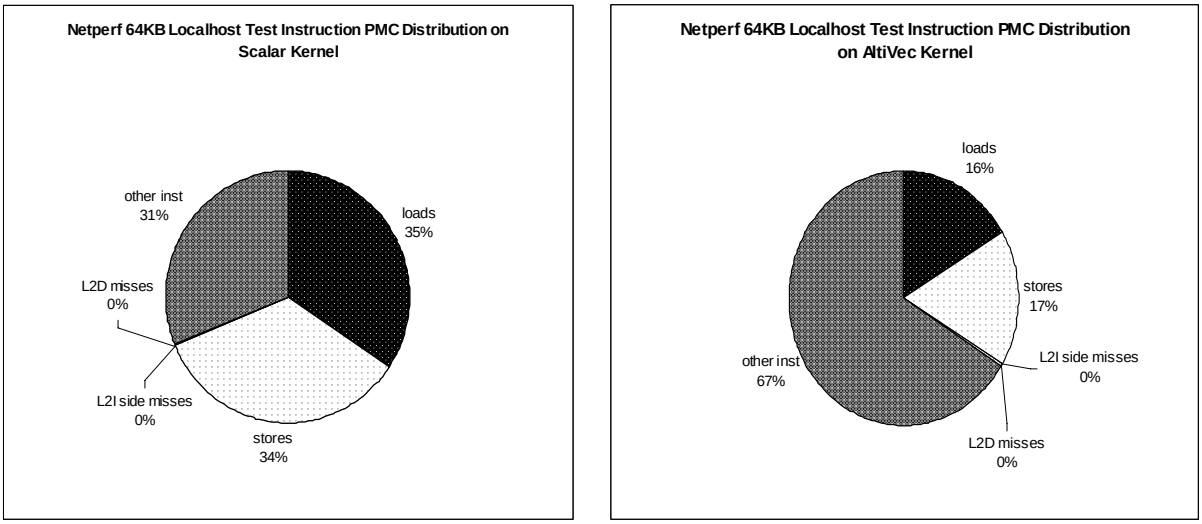



**Figure 12. Netperf/Netserver TCP Stream (Loopback) Test Performance Monitoring Counts**

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.

8. Netperf and Netserver are run on the same host system, consuming nearly 100% CPU.

# 4 Conclusion

The AltiVec unit provides Freescale PowerPC G4 processors with extra processing power. This application note demonstrates major performance enhancement contributed by using a few AltiVec-enabled library functions in the operating system protocol stacks. With a properly tuned buffer size, an AltiVec-enhanced protocol stack and socket layer can use half the CPU time to deliver the same TCP stream throughput compared to the original kernel or deliver more throughput using a simple protocol such as UDP.

This application note also explains and introduces common techniques for enabling AltiVec in multitasking operating systems. It allows the user to choose the best option to minimize the overhead of saving and restoring vector registers. Moreover, the independent approach provides an option to use AltiVec on an RTOS without AltiVec context support. The networking benchmark example demonstrates that kernel overhead can be minimized with an appropriately selected context management scheme.

In summary, AltiVec can provide major performance enhancement to applications in which large data blocks or multiple data streams are processed in parallel. Its speedup can be extended beyond the networking example described in this document to any application with similar parameters. Example applications are provided on the AltiVec website: simdtech.org.

# 5 Appendix

AltiVec function prolog and epilog for independent VR context management:

```
/* load/store VRs with index update by 16B */

#define STVXU(vr, ra, rb) .long\

         (31<<26)+((vr)<<21)+((ra)<<16)+((rb)<<11)+(231<<1) ;\

                                    addi rb, rb, 16; \


#define LVXU(vr, ra, rb)  .long\

(31<<26)+((vr)<<21)+((ra)<<16)+((rb)<<11)+(103<<1); \

                                 addi rb, rb, 16; \


/* This macro performs the push operation of the AltiVec stack. It does not have
 * to be atomic since the stack frame is incremented by 32 * 16B each time.
 * Everything will be restored if preemption occurs in the middle

 */

#define PUSH_ALL_VR(RP, RC)\

         lis      RP, altivecStack@h ; \

         ori      RP, RP, altivecStack@l ; \

         addi     RC, RC, -2; /* zero base, RC has ben incremented */ \

         rlwinm   RC, RC, 9, 0, 31 ; /* find the end of the stack 16B*32*RC */\

         add      RP, RP, RC; \

         STVXU    (0, 0, RP); \
```

```
        STVXU    (1, 0, RP); \
        STVXU    (2, 0, RP); \
        STVXU    (3, 0, RP); \
        STVXU    (4, 0, RP); \
        STVXU    (5, 0, RP); \
        STVXU    (6, 0, RP); \
        STVXU    (7, 0, RP); \
        STVXU    (8, 0, RP); \
        STVXU    (9, 0, RP); \
        STVXU    (10, 0, RP); \
        STVXU    (11, 0, RP); \
        STVXU    (12, 0, RP); \
        STVXU    (13, 0, RP); \
        STVXU    (14, 0, RP); \
        STVXU    (15, 0, RP); \
        STVXU    (16, 0, RP); \
        STVXU    (17, 0, RP); \
        STVXU    (18, 0, RP); \
        STVXU    (19, 0, RP); \
        STVXU    (20, 0, RP); \
        STVXU    (21, 0, RP); \
        STVXU    (22, 0, RP); \
        STVXU    (23, 0, RP); \
        STVXU    (24, 0, RP); \
        STVXU    (25, 0, RP); \
        STVXU    (26, 0, RP); \
        STVXU    (27, 0, RP); \
        STVXU    (28, 0, RP); \
        STVXU    (29, 0, RP); \
        STVXU    (30, 0, RP); \
        STVXU    (31, 0, RP); \

#define POP_ALL_VR(RP, RC)\
        lis      RP, altivecStack@h ; \
        ori      RP, RP, altivecStack@l ; \
```

```
addi     RC, RC, -1; /* zero base */ \
rlwinm   RC, RC, 9, 0, 31 ; /* find the end of the stack 16B*RC */\
add      RP, RP, RC; \
LVXU     (0, 0, RP); \
LVXU     (1, 0, RP); \
LVXU     (2, 0, RP); \
LVXU     (3, 0, RP); \
LVXU     (4, 0, RP); \
LVXU     (5, 0, RP); \
LVXU     (6, 0, RP); \
LVXU     (7, 0, RP); \
LVXU     (8, 0, RP); \
LVXU     (9, 0, RP); \
LVXU     (10, 0, RP); \
LVXU     (11, 0, RP); \
LVXU     (12, 0, RP); \
LVXU     (13, 0, RP); \
LVXU     (14, 0, RP); \
LVXU     (15, 0, RP); \
LVXU     (16, 0, RP); \
LVXU     (17, 0, RP); \
LVXU     (18, 0, RP); \
LVXU     (19, 0, RP); \
LVXU     (20, 0, RP); \
LVXU     (21, 0, RP); \
LVXU     (22, 0, RP); \
LVXU     (23, 0, RP); \
LVXU     (24, 0, RP); \
LVXU     (25, 0, RP); \
LVXU     (26, 0, RP); \
LVXU     (27, 0, RP); \
LVXU     (28, 0, RP); \
LVXU     (29, 0, RP); \
LVXU     (30, 0, RP); \
LVXU     (31, 0, RP); \
```

```
#define INC_VEC_USAGE(RC, RP)\
500:                                        \
        lis     RP, altivecUsage@h; \
        ori     RP, RP, altivecUsage@l;\
        lwarx   RC, 0, RP;          \
        addi    RC, RC, 1;          \
        stwcx.  RC, 0, RP;          \
        bne-    500b;                       \
       cmpi    cr6, 0, RC, 1;          \
       beq     cr6, 502f;              \
501:   \
       PUSH_ALL_VR(RP, RC)\
        lis     RP, altivecStackSize@h;     \
       ori     RP, RP, altivecStackSize@l; \
       lwz     RC, 0(RP);              \
       addi    RC, RC, 32;              \
       stw     RC, 0(RP);              \
502:   nop                             \

#define DEC_VEC_USAGE(RC, RP)\
600:                                        \
        lis     RP, altivecUsage@h; \
        ori     RP, RP, altivecUsage@l;\
        lwarx   RC, 0, RP;          \
        addi    RC, RC, -1;         \
        stwcx.  RC, 0, RP;          \
        bne-    600b;                       \
       cmpi    cr6, 0, RC, 0;          \
       beq     cr6, 602f;  /* no stack op */ \
601:   \
       POP_ALL_VR(RP, RC)\
        lis     RP, altivecStackSize@h;     \
       ori     RP, RP, altivecStackSize@l; \
       lwz     RC, 0(RP);              \
```

```
        addi    RC, RC, -32;              \
        stw     RC, 0(RP);               \
602:    nop               ;               \
```

Usage INC_VEC_USAGE(r6,r7)

Example code for using performance monitoring counters in Linux, compiled by GNU C compiler[9]:

```
/*****************************************************************************
 *     Copyright Freescale, Inc. 1989-2003 ALL RIGHTS RESERVED
 *
 *  $ID:$
 *
* You are hereby granted a copyright license to use, modify, and
* distribute the SOFTWARE, solely in conjunction with the development
* and marketing of your products which use and incorporate microprocessors
* which implement the PowerPC(TM) architecture manufactured by
* Freescale and provided you comply with all of the following restrictions
* i) this entire notice is retained without alteration in any
* modified and/or redistributed versions, and
* ii) that such modified versions are clearly identified as such.
* No licenses are granted by implication, estoppel or
* otherwise under any patents or trademarks of Freescale, Inc.
*
 * The SOFTWARE is provided on an "AS IS" basis and without warranty. To
* the maximum extent permitted by applicable law, Freescale DISCLAIMS ALL
* WARRANTIES WHETHER EXPRESS OR IMPLIED, INCLUDING IMPLIED WARRANTIES OF
* MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND ANY WARRANTY
* AGAINST INFRINGEMENT WITH REGARD TO THE SOFTWARE
* (INCLUDING ANY MODIFIED VERSIONS THEREOF) AND ANY ACCOMPANYING
* WRITTEN MATERIALS.
*
* To the maximum extent permitted by applicable law, IN NO EVENT SHALL
* Freescale BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING WITHOUT
* LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS
* INTERRUPTION, LOSS OF BUSINESS INFORMATION,
* OR OTHER PECUNIARY LOSS) ARISING OF THE USE OR INABILITY TO USE THE
* SOFTWARE.
* Freescale assumes no responsibility for the maintenance and support of
* the SOFTWARE.
 *****************************************************************************/
#include<stdio.h>
#include <linux/config.h>
#include <linux/module.h>
```

9. ANSI uses “__asm__” for inline assembly function whereas GCC uses “asm”.

```
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/compiler.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/init.h>
#include <linux/ioport.h>

#define MPC74XX_UMMCR0 936
#define MPC74XX_UPMC1 937
#define MPC74XX_UPMC2 938
#define MMCR2 944
#define BAMR  951
#define MMCR0 952
#define PMC1 953
#define PMC2 954
#define SIAR 955
#define MMCR1 956
#define PMC3 957
#define PMC4 958
#define MMCR0_PMC1SEL_LEFT_SHIFT_BITS (31-25)
#define MMCR0_PMC2SEL_LEFT_SHIFT_BITS (0)
#define MMCR1_PMC3SEL_LEFT_SHIFT_BITS (31-4)
#define MMCR1_PMC4SEL_LEFT_SHIFT_BITS (31-9)
/* PMC1 */
#define MMCR0_PMC1SEL_CYCLES (1)
#define MMCR0_PMC1SEL_INST  (2)
#define MMCR0_PMC1SEL_INST_DISP (4)
#define MMCR0_PMC1SEL_VSIU  (8)
#define MMCR0_PMC1SEL_LOAD_MISS  (11)
#define MMCR0_PMC1SEL_DISP_STALL  (13)
#define MMCR0_PMC1SEL_STORE  (21)
#define MMCR0_PMC1SEL_L1HITS  (24)
#define MMCR0_PMC1SEL_IL1_MISS  (32)
#define MMCR0_PMC1SEL_L2ISIDE_MISS (34)
```

```
#define MMCR0_PMC1SEL_STALL_BIU (37)
#define MMCR0_PMC1SEL_ALTIVEC_LOAD (48)
/* PMC2 */
#define MMCR0_PMC2SEL_CYCLES (1)
#define MMCR0_PMC2SEL_INST (2)
#define MMCR0_PMC2SEL_ITLB_MISS (6)
#define MMCR0_PMC2SEL_LOAD (11)
#define MMCR0_PMC2SEL_L1D_MISS (17)
#define MMCR0_PMC2SEL_IL1_RELOAD (25)
#define MMCR0_PMC2SEL_L2D_MISS (26)
#define MMCR0_PMC2SEL_SNOOP_HIT (37)
/* PMC3 */
#define MMCR1_PMC3SEL_CYCLES (1)
#define MMCR1_PMC3SEL_INST (2)
#define MMCR1_PMC3SEL_SNOOP_BUS (26)
#define MMCR1_PMC3SEL_L2SNOOP_HITS (27)
#define MMCR1_PMC3SEL_STALL_LRCTR (15)
/* PMC4 */
#define MMCR1_PMC4SEL_CYCLES (1)
#define MMCR1_PMC4SEL_INST (2)
#define MMCR1_PMC4SEL_STALL_FETCH (14)
#define MSR_PMM ((31-29)<<0x01)
#define DRV_MODULE_NAME"JPAN MPC7400 PMM TEST"
#define DRV_MODULE_VERSION"0.99"
#define DRV_MODULE_RELDATE"Fev 12, 2003"
#define KERN_INFO "<6>"
MODULE_AUTHOR("Jacob Pan);
MODULE_DESCRIPTION("Kernel module to access performance monitors");
MODULE_LICENSE("Freescale General Business Use");
typedef struct sPmcEvent
{
  int pmc_number;
  int event_number;
  const char name[80];
} tPmcEvent;
```

```
/* define a  test matrix */
tPmcEvent pmcEvent[] =
  {
    { 1, MMCR0_PMC1SEL_CYCLES, "Processor Cycles "},
    { 1, MMCR0_PMC1SEL_L1HITS, "L1 Cache hits "},
    { 1, MMCR0_PMC1SEL_INST_DISP, "Instruction Dispatched "},
    { 1, MMCR0_PMC1SEL_IL1_MISS, "L1 I-Cache Fetch Misses "},
    { 1, MMCR0_PMC1SEL_L2ISIDE_MISS, "L2 I-Cache Side Misses "},
    { 1, MMCR0_PMC1SEL_LOAD_MISS, "Load Miss Latency over threshhold "},
    { 1, MMCR0_PMC1SEL_STORE, "Stores Completed "},
    { 1, MMCR0_PMC1SEL_ALTIVEC_LOAD, "AltiVec Loads "},
    { 1, MMCR0_PMC1SEL_DISP_STALL, "Dispatch Stalls Due to Spec. Branch "},
    { 1, MMCR0_PMC1SEL_STALL_BIU, "Stalls BIU "},

    { 2, 1, "Processor Cycles "},
    { 2, 2, "Inst Completed "},
    { 2, MMCR0_PMC2SEL_ITLB_MISS, "I-TLB MIsses, no translation "},
    { 2, MMCR0_PMC2SEL_IL1_RELOAD, "L1 I-Cache Block Reloads "},
    { 2, MMCR0_PMC2SEL_LOAD, "Loads Completed"},
    { 2, MMCR0_PMC2SEL_L2D_MISS, "L2D Misses"},
    { 2, MMCR0_PMC2SEL_L1D_MISS, "L1D Misses"},
    { 2, MMCR0_PMC2SEL_SNOOP_HIT, "Snoop Hits"},

    { 3, 1, "Processor Cycles "},
    { 3, MMCR1_PMC3SEL_INST, "Inst. Completed "},
    { 3, MMCR1_PMC3SEL_STALL_LRCTR, "Stall on LR|CTR "},
    { 3, 26, "Bus Snoops "},
    { 3, 27, "L2 Snoop Hits "},

    { 4, 1, "Processor Cycles"},
    { 4, 2, "Instructions Completed"},
    { 4, 14, "Stalled Cycles Br. Fetch"},
    {-1, 0, "the end"}
  };
```

```
int pmc_read()
{
  unsigned long ummcr0=0xDEADBEEF, upmc1, upmc2=0;
  unsigned long ummcr1=0xDEADBEEF, upmc3, upmc4=0;
  unsigned long msr =0;
  asm volatile ("mfmsr %0 \n\t"
                : "=r" (msr));

  asm volatile ("mfspr %0, 936\n\t"
                "mfspr %1, 937\n\t"
                "mfspr %2, 938\n\t"
                : "=r" (ummcr0), "=r" (upmc1), "=r" (upmc2));

  asm volatile ("mfspr %0, 940\n\t"
                "mfspr %1, 941\n\t"
                "mfspr %2, 942\n\t"
                : "=r" (ummcr1), "=r" (upmc3), "=r" (upmc4));
  printk(KERN_INFO "%d\n%d\n%d\n%d\n",\
         upmc1, upmc2, upmc3, upmc4 );
  return 0;
}
static int
print_mmcr_event(unsigned long mmcr0, unsigned long mmcr1)
{
  int event[4];
  int i, j;
  event[0] = (mmcr0 >>6)&0x0000007F;
  event[1] = mmcr0 & 0x0000003F;
  event[2] = mmcr1 >>27;
  event[3] = (mmcr1 >>22)&0x0000001F;

  //  printk("Seleted PMM counts are \n");
  for(i=0; i<=3; i++)
    {
      printk("PMC%d:", i+1);
```

```
      j=0;
      while(pmcEvent[j].pmc_number != -1)
         {
           if((pmcEvent[j].pmc_number) == (i+1) && \
              (pmcEvent[j].event_number== event[i]))
             printk("%s(%d)",pmcEvent[j].name, event[i]);
           j++;
         }
      printk("\n");
    }
    return -1;
}
static int __init pmm_init(void)
{
  unsigned long msr =0, mmcr0=0, mmcr1=0;
  unsigned long mmcr0_mask=0, mmcr1_mask=0;

  /*   Set test case mmcrX selection */
  mmcr0_mask = (MMCR0_PMC1SEL_DISP_STALL<<MMCR0_PMC1SEL_LEFT_SHIFT_BITS ) \
    |(MMCR0_PMC2SEL_INST<<MMCR0_PMC2SEL_LEFT_SHIFT_BITS);

  mmcr1_mask = (MMCR1_PMC3SEL_STALL_LRCTR<<MMCR1_PMC3SEL_LEFT_SHIFT_BITS ) \
    |(MMCR1_PMC4SEL_CYCLES<<MMCR1_PMC4SEL_LEFT_SHIFT_BITS);

  print_mmcr_event(mmcr0_mask, mmcr1_mask);
  printk("-- MPC74XX Performance Monitoring --\n");
  asm volatile ("mfmsr %0 \n\t"
                : "=r" (msr));
  msr |= MSR_PMM;
  asm volatile ("mtmsr %0 \n\t"
                : "=r" (msr));
  asm volatile ("mfmsr %0 \n\t"
                : "=r" (msr));
  printk(KERN_INFO "NOW MSR = 0x%08X\n",  msr);
  /*   select events for PMC 1 to 2 controlled by MMCR0  */
```

```
  asm volatile ("mfspr %0, 952 \n\t"
                : "=r" (mmcr0));
  mmcr0|=mmcr0_mask;
  asm volatile ("mtspr %0, %1 \n\t"
                :: "i"(MMCR0),"r" (mmcr0));

  /*   select events for PMC 3 to 4 controlled by MMCR0  */
  asm volatile ("mfspr %0, 956 \n\t"
                : "=r" (mmcr1));
  mmcr1|=mmcr1_mask;
  asm volatile ("mtspr %0, %1 \n\t"
                :: "i"(MMCR1),"r" (mmcr1));
  return 0;
}
static void __exit pmm_cleanup(void)
{
/* read pmc before exit */
  pmc_read();
  asm volatile ("mtspr %0, %3 \n\t"
                "mtspr %1, %3 \n\t"
                "mtspr %2, %3 \n\t         "
                :: "i" (MMCR0), "i" (PMC1), "i" (PMC2), "r" (0));
  asm volatile ("mtspr %0, %3 \n\t"
                "mtspr %1, %3 \n\t"
                "mtspr %2, %3 \n\t         "
                :: "i" (MMCR1), "i" (PMC3), "i" (PMC4), "r" (0));

  printk(KERN_INFO "*********** PMM Cleanup ***********\n");
}

module_init(pmm_init);
module_exit(pmm_cleanup);
```

# 6 References

[1] Gary R. Wright, and W. R. Stevens “TCP/IP Illustrated Volume 2” 1993.

# 7 Document Revision History

Table 2 provides a revision history for this application note.

**Table 2. Document Revision History**

| Revision Number | Change(s) |
|---|---|
| 0 | Initial release |
| 1 | Updated the document in the Freescale template. No substantive changes. |

THIS PAGE INTENTIONALLY LEFT BLANK

***How to Reach Us:***

**Home Page:**
www.freescale.com

**email:**
support@freescale.com

**USA/Europe or Locations Not Listed:**
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
1-800-521-6274
480-768-2130
support@freescale.com

**Europe, Middle East, and Africa:**
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

**Japan:**
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064, Japan
0120 191014
+81 3 5437 9125
support.japan@freescale.com

**Asia/Pacific:**
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

**For Literature Requests Only:**
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com



Document Number: AN2581
Rev. 1
07/2006

freescale™ semiconductor