544

AT893-0087 WPB
S. T. Pan and G. Y. Tsao

IBM® Technical Disclosure Bulletin Vol. 37 No. 02A February 1994

Transmission Control Protocol/Internet Protocol Checksum Improvement for AIX 3.2 under RISC System/6000

The checksum, used in the Internet Protocol, treats the byte streams (either header or data) as a sequence of 16-bit integers, and defines the checksum to be the ones complement of the sum of all 16-bit intergers. Also, the sum and complement are defined to use ones complement arithmetic. Since the checksum is such an important procedure, AIX 3.2* routine "in_cksum" uses assembly language to code the calculation and wants to improve the performance as much as possible.

For current AIX 3.2 product, "in_cksum" calculation can be improved further (in terms of cpu cycles) if using instructions which take advantage of RISC/6000* architecture. It is estimated that about 18% performance improvement is achievable with the new approach proposed in this invention.

CURRENT AIX 3.2 COD :

The current main body of checksum routine "in_cksum" is as follows:

```
# Register Usage:
#       r5 = checksum
#       r6 = mbuf dynamic address pointer
#       r7 = mbuf dynamic counter
# Assume word alignment

        ....
sum_fast64:
      cmpi    0,r7,64             # while ( mlen > = 64 )
      blt     sum_fast16          # branch if not
#     go to here if at least 64 more bytes to be checksummed
#
sum_64loop:
      lm      r16, 0(r6)          # get next 64 bytes into r16 thru r31
@@@   ai      r7,r7,-64           # mlen -= 64
@@@   ai      r6,r6,64            # ptr += 64; CLEARS CO bit
#     DO 16 bit ones complemnt sum 32 bits at a time
#     by adding the shorts side by side we apeed up the loop considerably
#     the carrys are folded back in by the "ae" instruction and by
#     carrys into the high short of the register. After we do these
#     32 bit summing, the sum is folded back to 16 bits

      ae      r5,r5,r16
      ae      r5,r5,r17
```

```
        ae      r5,r5,r18
        ae      r5,r5,r19
        ae      r5,r5,r20
        ae      r5,r5,r21
        ae      r5,r5,r22
        ae      r5,r5,r23
        ae      r5,r5,r24
        ae      r5,r5,r25
        ae      r5,r5,r26
        ae      r5,r5,r27
        ae      r5,r5,r28
        ae      r5,r5,r29
        ae      r5,r5,r30
        ae      r5,r5,r31
@@@     aze     r5,r5           # fold in carry bit!
#
@@@     cmpi    0,r7,64         # END while ( mlen >=64 )
@@@     bge     sum_64loop      #
```

The current implementation of the TCP/IP checksum as shown in the loop "sum_fast64" has many drawbacks:

1. The carry-bit-reset, few instructions from the beginning of the loop, (ai r6,r6,64) and the carry-bit-folding, few instructions before the end of the loop, (aze r5,r5) are unnecessary. For implementing the 16-bit 1's complement checksum, it is only necessary to have the carry bit cleared before the start and folded in after the end of the checksum computation.
2. The ONLY advantage of using the load multiple instruction "lm" is to save few instruction cache spaces. The fixed-point unit (FXU) consumes the same amount of cycles to process it as if it were using multiple single-word load instructions instead. For example, for the "lm" instruction coded in this loop, 16 CPU cycles are required (assuming no data cache misses during the load operations).
3. The worst problem in this implementation is the use of a compare instruction immediately followed by a branch instruction based on the compare result. Due to the hardware implementation of of the Power architecture, there's a "3-cycle pipeline bubble" guaranteed in between the executions of the "cmpi" and the next valid instructions. This pipeline bubble is basically contributed by the cycles required to transfer the condition register result to the ICU, flush the FXU pipeline, and dispatch the target instructions to the FXU.
4. In the absence of data cache misses, a total of 39 CPU cycles is required to finish the checksum computation of a 64-byte package.

NEW CODE:

```
#
# Prepare fast 8-byte checksum, using ctr register as branch control
# This is only performed once !!
#
```

```
        ai      r8,r7,0          # move byte counter in r7 to r8
        andil.  r7,r7,0x7          # r7 = remaining bytes ( < 8)
        sri.    r8,r8,3          # r8 = r8 > > 3
        bz      sum_fast2          # handle checksum < 8 bytes
        mtctr   r8               # move r8 into ctr register
        ai      r6,r6,-4         # prepare to use "load with update"
                                 # by backing-off the mbuf pointer.
                                 # this also resets the CA bit.
                                 #
#
# fast 8-byte loop (main body)
#
sum_8loop:
        lu r16,  4(r6)           # load 1st word & update the mbuf
pointer
        lu r17,  4(r6)           # load 2nd word & update the mbuf
pointer
                                 # we DONT need a separate
                                 # instruction to update r6!
        ae      r5,r5,r16          # perform checksum
        ae      r5,r5,r17          # perform checksum
        bdn     sum_8loop            # zero-cycle branch !
        aze     r5, r5           # fold in carry bit!
#
# Handle checksum < 8 bytes
#
sum_fast2:
        ....
```

HOW ARE BENEFITS DELIVERED?

1. Use less memory - There are only 5 instructions in the main loop " sum_8loop", whereas in "sum_64loop", there are 22 instructions.
2. Use fewer instructions or cycles
   a. The byte count (r7) does not need to be updated every 64 bytes, since we are using the Counter register. It decrements automatically (bdn).
   b. The mbuf address pointer (r6) does not need to be updated every 64 bytes, since we are using "Load with Update" instruction (lu). Lu automatically updates the address pointer whenever it loads a new word.
   c. The loop closing branch in our approach has zero-cycle delay (as opposed to 3-cycle delay in the current implementation). This is because that the branch-on-count instruction "bdn" can be resolved deterministically (it is independent of the FXU operatio n) and the hardware can prefetch the target instructions and dispatch them based on the outcome of the pre-resolved branch.

d. The new code does not need the "aze" instruction in the loop while current code needs it. in the 64-byte loop.

(Notice @@@ marks in the current code are not in the present code)

The benefits are listed below:

- For a 64-byte packet checksum calculation:
- Current code take 39 cycles
- New code take 32 cycles
- Net performance improvement = 18%

* Trademark of IBM Corp.