Memory Bandwidth of the Power2 Architecture

King Lee [1]

Report RND-94-013 September 1994

NAS Systems Division

NASA Ames Research Center, Mail Stop 258-6

Moffett Field, CA 94035 – 1000

August 8, 1994

[1]The mailing address for Lee is Computer Science Dept., California State University, Bakersfield, CA 93309. His e-mail address is klee@nas.nasa.gov.

Memory Bandwidth of the POWER2 Architecture

Abstract

The POWER2 is a new implementation of the POWER architecture that has peak performance and memory bandwidth comparable to a single processor Cray YMP. This report examines several loops that were chosen primarily to investigate the memory system. A indication of the effective memory performance can give grounds for predicting the performance of different codes. This report presents measurements of memory bandwidth under different conditions and makes suggestions on how the compiler might make further optimizations.

1 Introduction

The rapid improvement in microprocessor technology has promised a cost effective way to solve scientific problems. Until recently, the floating point performance of RISC microprocessors was disappointing when applied to large scientific problems. For example, the Intel i860^{TM} microprocessor achieved only about 10 million floating point operations (MFLOPS) out of a theoretical peak performance of 60 MFLOPS on a DAXPY when the operands were not in cache [3]. On many codes the limitation on performance is memory bandwidth. The POWER2 , a second generation implementation in the the RS6000 architecture, significantly improved floating point performance and memory bandwidth so that it can effectively handle problems that were previously solved only by supercomputers.

The POWER2 has two versions; a high performance version and a lower cost version. This report is concerned with the high performance version on an IBM 590 workstation. The high performance POWER2 has a 256K 4–way set associative data cache with 256 byte (32 double precision word) lines. The low cost design has a 128K 4–way set associative cache and a line size of 128 bytes. The POWER2 has a 4K page size. To speed up paging translation, the CPU contains a 512–entry two–way set–associative translation lookaside buffer (TLB). Some of the more important features of the high performance POWER2 version that contribute to the high floating point performance are:

- The ability to dispatch two fixed point and two floating point instructions every clock. Each floating point instruction can perform two floating point operations, so it is possible to execute four floating operations each clock.

- Quadload and quadstore instructions that load or store two adjacent double precision registers. The ability to load two operands in one clock is important for vector operations. For example, the DAXPY loop requires 3 memory references for every two floating point operations. Without the quadload instruction, we would require 3 clocks to load or store the operands. With the quadload and quadstore instructions, we require an average of 1.5 clocks to load and store the operands.

- The ability of the memory system to have two pending operations [5] This will help hide some of the latency when loading a line of cache.

- A 32 byte wide bus that can deliver 32 bytes every clock.

With a clock of 67.5 MHz, the peak performance is 270 MFLOPS. The maximum bandwidth between memory and the CPU is over 250 million words per second (MWDS) or 2 billion bytes per second (GBS). A single processor of the Cray Y–MP had a peak performance of 330 MFLOPS and a memory bandwidth of (assuming 3 ports to memory) of 480 MWDS. The high performance does not come cheaply. The POWER2 implementation is a 6 chip set, with a total of 23 million transistors. Most of the current single chip microprocessors have fewer than 5 million transistors.

2 Performance of Memory Loads and Stores

This section presents some performance measurements of the POWER2 microprocessor. The loops were compiled with the highest level of optimization on version 3.01 of the FORTRAN compiler. The flags used were

```
-O3 -qhot -qarch=pwr2 -qtune.
```

Timing measurements were made during the day when there were other users on the system. In order to disregard the effects of other users, each loop was timed 200 times. The minimum time was used to compute the performance. The timing of each iteration of the loops generally ranged from 15 microseconds to 40 microseconds and the resolution of the clock was about 1 microsecond. Therefore the uncertainty due to the resolution is about 3 to 6 per cent or more depending on the time of the loop. Measurements were made over a number of lengths and the reported measurements are reported for length vector length 2000. This was a large enough vector length to reach asymptotic performance and still stay in cache at stride 16. The vectors were placed in common to align them on quadword boundaries. No two vectors would have their first element on the same cache line.

The first series of loops are intended to measure memory bandwidth of loads from cache and memory. We tried to use a "do nothing" loop to measure performance, but the optimizing compiler gave erratic results. The first loop was essentially:

```
          do 100 i = 1, n
100          s = s + x(i)
```

LOOP 1

To measure the performance of cache loads, the cache was loaded with the vector x before performance was measured. To measure the performance of memory loads, the cache was loaded with a scratch vector before performance was measured. The theoretical peak performance of this loop is 136 MFLOPS and was limited by the floating point operations.

Initial measurements of LOOP 1 were disappointing even though an examination of the assembler listing showed that the loop was unrolled. Manually unrolling the loop improved performance dramatically. The reason is that manually unrolling the loop caused more registers to be used to accumulate sums. The assembler list showed that quadloads were used. When a quadload instruction is executed with unaligned operands, the tag directory may have to be accessed twice if the quadload crosses a cache line. The double access may cause a stall. Therefore tests were performed to see if alignment would affect performance. The results are given in TABLE 1.

Vector Sum (MFLOPS) Unit Stride								
	Depth of Unrolling							
	1	2	3	4	5	6	7	8
Aligned								
Cache Load	31.7	59.9	80.3	107.5	111.1	134.1	131.1	131.1
Mem Load	32.0	58.5	64.0	77.0	75.2	78.4	75.2	78.4
Unaligned								
Cache Load	31.7	59.9	80.3	107.5	111.1	123.4	131.1	131.1
Mem Load	31.9	56.7	61.5	75.9	74.6	73.3	76.3	77.0

TABLE 1

Each FLOP requires a memory reference of 8 bytes. The best memory performance was over 1 GBS when data was in cache and the loop was manually unrolled 8 times. When the data was in memory the best performance was over 600 MBS. On a cache miss there is 15 to 20 clock latency [1] before data appears in cache. After the latency, the cache line should be filled in 8 clocks. *Assuming* a latency of 15 clocks and a cache fill time of 8 clocks, the time it takes to fill a 256 byte line would be 23 clocks, or 700 MBS, which is consistent with what we measured. The peak performance to load from memory may be higher if we can overlap some of the latency of two back–to–back cache loads.

Manually unrolling this loop can improve performance by a factor of 4 when data is in cache, and a factor of 2 when data is in memory. An examination of the assembler listing of LOOP 1 and the manually unrolled loops showed that the latter cases used more registers to hold intermediate values. Evidently stalls were introduced when one or two registers were used to accumulate the sum. The performance for this loop, designed to measure memory performance, may have been limited by floating point operations. We note that the difference between aligned and unaligned data was relatively small in this case.

Next we considered the case when we had arbitrary stride incx. The loops are equivalent to the following code fragment for the unrolled loop:

```
          do 100 i = 1, 1+(n-1) * incx,incx
100           s = s + x(i)
```

Loop 2

When stride is not one, then there is no need to use the quadload instruction. If the compiler does not know the stride at compile time, as is the case when the stride is variable, the compiler generates doubleload instructions. The instruction set allows us to load a word and update a pointer with one instruction so that each fixed point unit should be able to issue a load every clock. Therefore, the POWER2 should be able to issue a doubleload instruction every clock. First, the loop was run when we forced the vector to be in cache. The results are given in Table 2.

Vector Sum (MFLOPS) Variable Stride – Cache Loads								
Stride	Depth of Unrolling							
	1	2	3	4	5	6	7	8
1	31.6	59.1	77.7	84.7	101.1	96.4	111.1	104.9
2	31.6	59.1	77.7	85.2	101.1	95.9	111.1	104.9
4	31.6	59.1	77.3	84.7	101.1	97.0	111.1	104.9
8	31.6	59.1	77.7	84.7	101.1	95.9	111.1	104.9
16	31.1	56.1	68.8	68.2	85.6	81.8	91.2.2	90.6

Table 2

The maximum performance is still over 800 MBS at stride 1, a 20 percent decrease in performance compared to Table 1. The reason for the decrease is because for variable stride we are using doubleload loads instead of quadload loads. The results of the last row (stride 16) are rather puzzling. The slight decreases in performance seem indicative of some cache misses. However, with a vector length 2000 and a stride of 16, there should be no cache misses. Further investigation showed that there was the slight performance degradation when the product of the stride and vector length was about 28000.

We next investigated the performance when the operands are from memory. Before the load was performed, the cache was filled with a scratch vector, forcing operands from LOOP 2 to come from memory. The results are given in TABLE 3.

Vector Sum (MFLOPS) Variable Stride – Memory Loads								
Stride	Depth of Unrolling							
	1	2	3	4	5	6	7	8
1	31.7	51.9	59.1	63.8	65.3	62.6	64.8	66.8
2	31.4	45.7	44.7	45.0	45.8	45.5	43.8	45.7
4	23.8	23.8	23.7	23.8	23.0	23.3	23.3	23.7
8	12.4	13.1	13.3	12.4	12.7	12.2	12.7	12.4
16	6.8	6.8	6.8	6.8	6.8	6.8	6.8	6.8
32	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4
64	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5

TABLE 3

The performance varied from over 520 MBS to 27 MBS depending on the stride. When we have variable stride (using doubleloads), there is a 15 percent degradation in performance compared to the case of unit stride (using quadloads) as can be seen by comparing the first row of TABLE 3 and the second row of TABLE 1. We expect the performance to be cut in half if we double the stride until we get to a stride of 32. At stride 2 we load a line of cache on every miss, but we use half of the data that we load. At stride 4 we also load a line of cache on every miss, but we use one fourth of the words loaded. At stride 32 and larger we use only one of the 32 words of data that we load on each cache miss. The difference in performance between the best case (600 MBS using quadloads) and the worst case (28 MBS for stride 32) is large. This is due to the relatively large latency, large line size, and high bandwidth.

At stride 32 the time to perform an operation should be very close to the time to load a line of cache. From the performance for stride 32 we infer that the time to load a cache line is about 20 clocks which would work out to about 800 MBS effective bandwidth from memory.

A second series of loops was designed to measure the performance of storing data. The first loop is given in LOOP 3 and does not involve any floating point operations.

```
          do 100 i = 1, n
100           x(i)=1.0
```

LOOP 3

The results are given in TABLE 4.

Stores (Words/Sec) Unit Stride								
	Depth of Unrolling							
	1	2	3	4	5	6	7	8
Aligned								
Cache Store	162.9	161.3	162.9	166.1	159.8	162.9	156.8	164.5
Mem Store	86.9	84.3	85.1	83.5	83.1	85.2	82.6	83.9
Unaligned								
Cache Store	137.5	137.5	116.8	139.8	135.3	137.5	132.1	129.8
Mem Store	78.0	77.7	75.6	79.9	77.7	78.0	75.2	79.9

TABLE 4

The performance of storing data is over 1.3 GBS into cache and 670 MBS into memory. The difference in performance between aligned and unaligned stores varies from 20 per cent when data is in cache to 5 per cent when data is not in cache. In this, and in the subsequent loops, unrolling loops did not always give significantly better performance. This is consistent with the conjecture for LOOP 1 that the necessity for unrolling was due to stalls caused by data dependencies on the registers. For the sake of brevity, data for unrolled loops are omitted.

We next considered the case where we allowed for non unit stride.

```
          do 100 i = 1, 1+(n-1) * incx,incx
100           x(i) = 1.0
```

LOOP 4

First we tested the stores into cache; the cache was loaded with the vector `x` before the above loop was executed. We expect that performance would decrease because we cannot use the quadstore instruction. The results are given in TABLE 5.

Stores (Words/Sec) Non–unit Stride to Cache								
Stride	Depth of Unrolling							
	1	2	3	4	5	6	7	8
1	80.3	79.5	77.7	79.5	78.4	77.7	77.0	79.5
2	80.3	79.5	77.7	79.5	78.4	77.7	77.0	79.5
4	80.3	79.5	77.7	79.5	78.4	77.7	77.0	79.5
8	80.7	79.5	77.7	79.5	78.4	77.7	77.0	79.5
16	70.8	70.8	68.8	70.5	70.5	68.8	67.7	70.5

TABLE 5

For stores into cache we can get 640 MBS and performance is not significantly affected by stride or manually unrolling the loop. Note that the performance with unit stride is about half that when we can use the quadstore instruction as in TABLE 4. One must have instructions like the quadstore to take advantage of the wide data paths.

Next, the cache was loaded with a scratch vector before the LOOP 4 was executed. The results are given in TABLE 6. The memory system has 8 memory banks and the measurements at various strides showed that performance was not adversely affected with power of 2 strides.

Stores (Words/Sec) Non–unit Stride to Memory								
Stride	Depth of Unrolling							
	1	2	3	4	5	6	7	8
1	55.8	54.6	54.3	54.3	54.3	54.3	53.6	54.6
2	42.0	40.3	39.9	41.8	41.8	40.1	41.4	42.0
4	23.8	23.8	23.7	23.8	23.8	23.8	23.7	23.8
8	12.5	12.4	12.4	12.4	12.4	12.4	12.4	12.4
16	6.9	6.9	6.8	6.9	6.9	6.9	6.9	6.9
32	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6
64	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3

TABLE 6

The performance of stores to memory varied from 420 MBS to 20 MBS, depending on stride. At stride 1 we have about 35 percent decrease in performance from using quadstores (see second row of TABLE 4). Again there is a very wide difference wide range in performance between the best case of stride 1 and the worst case of stride 32.

3 DCOPY and DAXPY Routines

In this section we present the performance of some slightly more complex loops. We start with the DCOPY loop which copies one vector to another. As before, we consider the case where the quadload and quadstore instructions can be used.

```
          do 100 i = 1, n
100          y(i)=x(i)
```

LOOP 5

As in the previous section the vector lengths in all cases were 2000. The vectors were aligned so that their first elements of the respective vectors would not occupy the same cache line. Measurements were also made with the loops manually unrolled. Generally the manually unrolled loops did not improve performance. This should be expected because there are no floating point operations and therefore there is less chance of register conflicts. For the sake of brevity, the measurements for the unrolled loops are omitted.

FORTRAN DCOPY (Words Copied per Second) Unit Stride				
	Both Aligned	Source Aligned	Target Aligned	Neither Aligned
Both vectors in cache	119.0	99.9	108.9	99.9
Source vector in cache	45.8	43.7	42.9	41.6
Target vector in cache	70.8	66.8	63.1	62.6
Both vectors in memory	45.0	44.1	42.4	41.5

TABLE 7

Each word copied requires a load and a store and is counted as moving 2 double precision words (16 bytes). Therefore the performance varies from over 1.9 GBS when both vectors are in cache, to 540 MBS when the vectors are in memory. The differences in performance between having both vectors aligned and both unaligned vary from 5 to 20 percent.

We next consider the case of variable stride for LOOP 6.

```
          j = 1
          k = 1
          do 100 i = 1, n
               y(k)=x(j)
               k = k + incy
100            j = j + incx
```

LOOP 6

In this loop we do not expect that the alignment would make a difference since quadload instructions are not generated.

FORTRAN DCOPY (Words Copied per Second) Variable Stride					
	incx = 1 incy = 1	incx = 4 incy = 4	incx = 8 incy = 8	incx = 16 incy = 16	incx = 32 incy = 32
Both vectors in cache	64.3	63.3	53.6	na	na
Source vector in cache	34.5	12.8	7.0	2.6	na
Target vector in cache	43.4	22.6	8.3	2.9	na
Neither vectors in cache	32.7	12.8	6.8	2.6	1.3

na: vector does not fit in cache for these strides

Table 8

The performance ranged from 1 GBS with unit stride, to 20 MBS when we had the worst case stride of 32. When data is in cache and the stride is 1 the performance using quadloads and quadstores (Table 7) is almost double that of using doubleloads and doublestores. Again, this points to the importance of the quadload and quadstore instruction.

The DCOPY routine is available in the ESSL library. We measured the performance of library routine by replacing Loop 6 with a call to the ESSL library DCOPY. The results are given in Table 9.

ESSL DCOPY(Words Copied per Second)					
	incx = 1 incy = 1	incx = 4 incy = 4	incx = 8 incy = 8	incx = 16 incy = 16	incx = 32 incy = 32
Both vectors in cache	128.1	33.0	31.4	na	na
Source vector in cache	45.2	11.8	6.7	2.3	na
Target vector in cache	69.3	19.7	8.0	2.9	na
Neither vector in cache	45.8	12.2	6.6	2.6	1.2

na: vector does not fit in cache for these strides

Table 9

The performance varies from 2 GBS (stride 1 in cache) to 20 MBS (stride 32 in memory). The first *column* of Table 9 is similar to the first *column* of

TABLE 7, leading us to conjecture that DCOPY was coded using quadloads and quadstores. However, the first *row* of TABLE 9 is significantly less than the first *row* of TABLE 8 when the stride is not 1. We conclude that using quadloads with non–unit stride, in DCOPY at least, does not give optimal results.

It might be interesting to consider the C routine bcopy. This routine copies bytes and requires stride 1.

bcopy(Words Copied per Second)				
	both in cache	source in cache	target in cache	neither in cache
Words/Sec	29.7	21.0	25.0	20.9

TABLE 10

The performance varies from 480 MBS when data is in cache to 320 MBS, significantly less than using DCOPY. If the bcopy routine loads the integer registers, the best than can do is load two integer registers per clock. Since the integer registers are 4 bytes wide, this would mean we are loading 8 bytes per clock, or 540 MBS is the best we can do.

We next investigate the DAXPY loop. As before, we first consider the case that can use quadload as in LOOP 7.

```
      do 100 i = 1, n
100      y(i)=y(i) + a * x(i)
```

LOOP 7

The results are given in TABLE 11.

FORTRAN DAXPY(MFLOPS) Unit Stride				
	Both Aligned	x Aligned	y Aligned	Neither Aligned
Both in cache	161.3	135.8	147.8	120.3
x in cache	78.6	73.7	76.6	71.1
y in cache	107.8	95.3	100.4	99.9
Neither in cache	78.5	73.1	75.9	71.0

TABLE 11

The performance varied from 161 MFLOPS to 71 MFLOPS. We have 3 operands for every 2 FLOPs. The performance is dominated by how fast we can load and store operands, since we can load or store 2 operands in one clock (for each fixed point unit) and complete 2 floating point operations (for each floating point unit). Assuming we are using quadloads, operands are in cache, and both fixed and floating point operation units are busy, one would expect 4 FLOPs every 1.5 clocks. This gives a best possible performance of 178 MFLOPS when the operands are in cache. Our compiled code comes surprisingly close to the best possible performance.

We next measured the performance for LOOP 8.

```
          j = 1
          k = 1
          do 100 i = 1, n
             y(k)=y(k)+a * x(j)
             k = k + incy
100          j = j + incx
```

LOOP 8

In this loop we do not expect that the alignment would make a difference since quadload instructions are not generated. There are instructions that load and update pointers in one clock. Therefore the updating the pointers should not affect the performance.

FORTRAN DAXPY (MFLOPS) Variable Stride					
	incx = 1 incy = 1	incx = 4 incy = 4	incx = 8 incy = 8	incx = 16 incy = 16	incx = 32 incy = 32
Both vectors in cache	89.0	88.3	80.6	na	na
x in cache	56.7	27.3	13.6	6.1	na
y in cache	69.3	42.2	14.4	6.2	na
Neither vector cache	57.4	27.2	13.7	6.0	3.0

na: vector does not fit in cache for these strides

TABLE 12

If we compare the first column of TABLE 12 with the first column of TABLE 11, we see that the using doubleloads instead of quadloads significantly degrades performance at stride 1. When using doubleloads, we will require 3 clocks for every 4 FLOPS instead of 1.5 clocks when using quadloads. The measurements are consistent with the calculations and reemphasize the importance of the quadload instruction.

We also measured the performance obtained from the ESSL library. The results are given in the following table.

Library DAXPY (MFLOPS)					
	incx = 1 incy = 1	incx = 4 incy = 4	incx = 8 incy = 8	incx = 16 incy = 16	incx = 32 incy = 32
Both vector in cache	168.6	78.9	69.7	na	na
x in cache	78.9	24.3	13.3	6.0	na
y in cache	109.2	39.5	15.1	6.5	na
Neither vector in cache	74.5	25.1	13.5	6.0	3.0

na: vector does not fit in cache for these strides

TABLE 13

Like the case of DCOPY, the first column in TABLE 13 is similar to the first row in TABLE 11 which indicates that the DAXPY is coded using quadloads and quadstores. With nonunit stride the performance of FORTRAN

DAXPY is over 10 per cent greater than the ESSL version which can be seen by comparing the first rows of TABLE 12 and TABLE 13.

4 Summary

The performance that users achieve from a computer depends on the hardware and the compiler. The compiler seems to generates good code since the performance of compiled DCOPY and DAXPY approached those of library subroutines. However we suggest the following improvements be made:

- The compiler should unroll loops like LOOP 1 to give close to optimal performance when using highest level of optimization. In TABLE 1 we saw a factor of 4 performance between an unrolled and rolled loop. While many scientific programmers are capable of unrolling loops, it is not always easy to determine the optimal depth of unrolling. Also code written for other machines may not be unrolled because it is not needed for those architectures. Those codes may not run well on the POWER2 .

- The library bcopy routine should make a runtime test on the vector length and make a jump to code that uses, or does not use, the quadload and quadstore instructions depending on vector length. It might be worth while to make runtime tests for length and stride for SCOPY, CCOPY, ZCOPY, SAXPY, etc. if they do not already do so.

- The library routine DCOPY should incorporate a run time test of the stride and jump to code that that uses, or does not use, the quadload and quadstore instructions depending on the stride. There should be a substantial improvement in performance when the data is in cache and the stride is not one. This optimization was used in [4]. The Cray compilers sometimes make a run time test on vector length, and use or do not use vector instructions based on the vector length.

What is notable about the POWER2 system is that not only does it have a high peak performance but that it can sustain a relatively high fraction of the peak performance. This is because the architecture is well balanced and the compiler is able to generate good code making use of some of the features of the POWER2 . While it may be premature to predict the performance on more complex loops, the sustained 600 MBS effective memory bandwidth to memory with stride 1 is very encouraging. The relatively poor effective memory bandwidth with large stride will mean that we must place greater emphasis blocking the algorithms.

In the past performance has increased because of improvements in speed and improvements in architecture. We can expect improvements in the speed for the POWER2 to continue, especially as they reduce the number of chips in the microprocessor. Improvements in architecture may be harder to implement. There was a great improvement in performance when going from a single floating point pipe to two floating point pipes *and* the quadload and quadstore instructions. We should not expect equal improvement in performance if we went to four floating point and fixed point pipes and octload (load 4 double precision words) instructions, other factors being equal. This is because it may be difficult to make a 4 accesses the TLB and cache tag tables each clock, and to unroll a loop to the optimal depth because of insufficient registers, Furthermore any stalls would for whatever reason would cause 4 pipes, instead of 2 pipes, to stall. The Cray avoided these problems with vector instructions.

ACKNOWLEDGEMENT: The author thanks Samuel Fineberg and William Saphir for their careful reading of this paper and their editorial comments.

References

[1] Ramesh Argarwal and Fred Gustavson. *Algorithm and Architecture Aspects of Producing ESSL BLAS on POWER2*, IBM Journal of Research and Development. To appear.

[2] Troy Hicks, Richard Fry, and Paul Harvey. *Power2 Fixed-Point, Data Cache, and Storage Control Units*, IBM Journal of Research and Development. To appear.

[3] King Lee. *On the Floating Point Performance of the i860*$^{\text{TM}}$ *Microprocessor*, RNR-90-019, October, 1990. NAS, Ames Research Center, Moffett Field, CA 94035.

[4] King Lee. *The Performance of the Intel i860XP(tm)*, RND-94-001, October, September, 1993. NAS, Ames Research Center, Moffett Field, CA 94035.

[5] D. J. Shippy, T. W. Griffith, and G. Braceras. *Power2 Fixed-Point, Data Cache, and Storage Control Units*, IBM Journal of Research and Development. To appear.

[6] Stephan White and Sudhir Dhawan *Power2: Next Generation of the RISC System/6000 Family*, IBM Journal of Research and Development. To appear.