Applied Numerical Mathematics 10 (1992) 59–72
North-Holland

APNUM 329

Influence of memory systems on vector processor performance

Dik T. Winter

Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, Netherlands

Abstract

Winter, D.T., Influence of memory systems on vector processor performance, Applied Numerical Mathematics 10 (1992) 59–72.

This paper describes the influence of memory systems on the speed of vector operations on vector processors. In particular, attention is focused on low-level vector operations rather than on complete programs. Experiments have been done on six different machines and the results are analyzed.

Keywords. Memory system, cache memories, performance analysis, vector processors.

1. Introduction

With the current technology very fast processors are available. Typically the fastest processors allow pipelined processing on a series of (array-) elements, where each operation may take multiple clock cycles to complete but where the pipelining allows for one result (or even more) in each clock cycle. To make full use of such a facility requires the ability to fetch the operands and store the results fast enough in some memory place. However, current memories are not fast enough to allow all those fetches and stores in a single clock cycle. A number of techniques are known to overcome this problem:

(a) *Use of vector registers*. This is an easy way to simplify memory contention. Normally only a single fetch or a single store is required during each clock cycle for filling the vector registers or for storing a vector register result. A number of machines allow for concurrent execution of a number of vector loads/stores; this requires multiported memory, which is typically more expensive.

(b) *Interleaved memory*. Memory is subdivided into a number of equally sized smaller memories (memory banks). The lower bits of an address name the memory bank that contains the location addressed by it. In the case of contiguous vectors, for a vector operand fetch/store, each bank gets a request only ever *n*th clock cycle where *n* is the number of memory banks.

(c) *Caches*. Caches are well known for scalar processors and their effect on those processors has been studied extensively, but their effect on vector processing is less well known. The cache is memory with a very high speed that is placed between main memory and the processor. A request for a datum from memory will go through the cache. If the cache already contains the datum, it is returned immediately; if it is not in cache, it is fetched from memory, passed on to the processor and stored in the cache. The reason this can give a large speed-up is because in general a datum once referenced by a program will be reused a number of times, and the second and later times there is no need to reference memory.

Of course multiple techniques can be combined, and we will see examples of such combinations.

In this report we discuss the effects of some strategies on vector operations based on examples. We are especially interested in nontrivial speed degradation for some strategies. In Section 2 we will discuss the effect of memory interleaving; in Section 3 we will discuss the effect of a cache. The effect of vector registers depends a little on other memory strategies used: this will be discussed in Section 4.

2. Machines with memory interleaving

Memory interleaving was designed with the purpose to speed up access of contiguous elements in memory. In general you will find that a processor with a clock cycle of K seconds will have memory attached with a cycle time of more than K seconds. Clearly it is not possible to request a datum every clock cycle from such memory, so memory is divided in banks. Although each memory bank has a cycle time larger than K, it is possible to fetch contiguous elements each clock cycle since they come from different banks. The number of banks should suffice to compensate for the difference between memory cycle time and processor cycle time; and indeed in general this is more than compensated, i.e. if memory cycle time is N seconds and processor cycle time is K seconds, the number of memory banks is generally chosen much larger than N/K.

There is an obvious effect if memory is accessed in a noncontiguous (equidistant) manner (e.g. in Fortran a row of a matrix), successive accesses may hit the same memory bank, leading to a severe degradation in performance. Indeed, also cases where the same memory bank is hit every other time, or even every fourth time may show a degradation of performance. There is no way around it; it is inherent in the design.

When accessing rows of a two-dimensional array in Fortran, this effect can be countered quite effectively by declaring all such arrays with a number of rows that is a power of two plus one (or indeed, any odd number would do). In that case accesses to successive elements of a row of a matrix will access different memory banks.

It has also been proposed to choose the number of memory banks a small prime number (like 13, 17, etc.). The benefits are clear; even with noncontiguous (but equidistant) accesses the probability of hitting the same memory bank with every access is quite small; and it is impossible to hit the same bank with every other access. Still, most systems use a power of two for the number of memory banks. The reason is that a power of two allows a division across memory banks based on the low-order bits of the address; in the case of a prime number a (more complicated) calculation has to be performed to get the correct bank and the address within the bank.

The technique of memory interleaving can be combined with vector registers or with cache. When combined with vector registers the effect mentioned above will be the only effect that can be seen. If combined with cache, memory interleaving will not be visible, except possibly for the memory bank conflicts indicated above, but this will be discussed in more detail in Section 3. However, there are machines that use memory interleaving and do not use vector registers or caches. On these machines a further effect can be found: it is then possible that the two operands and the result of a vector operation do lead to bank conflicts.

We will study the afore-mentioned effects for the following Fortran loop:

```
      DO 30 II = 0, NBANKS − 1
        DO 20 JJ = 0, NBANKS − 1
          T = GETTIME()
          DO 10 I = 1, 512
            A(I) = B(I + II) + C(I + JJ)
10        CONTINUE
          T = GETTIME() − T
          . . .
20      CONTINUE
30    CONTINUE
```

where A, B and C are declared (using COMMON) such that A(1), B(1) and C(1) are 512 elements apart. A similar test was performed for the multiplication of vector elements. Clearly for II = 0 and JJ = 0 the operands for a single operation come from the same memory bank and the result goes into the same bank. Varying II and JJ forces the time intervals between accesses to the same memory bank to vary. A priori we cannot expect the case II = 0, JJ = 0 to be the worst case as storing the result will occur a few cycles later than fetching the operands.

We have analyzed the results for this loop on two vector processors.

2.1. CDC Cyber 205 [3]

The number of memory banks on the CDC Cyber 205 is model-dependent, but it is at least 64. Most vector operations on the CDC Cyber 205 take two memory vector operands and return one memory vector result. As memory cycle time is four times the processor cycle time, the probability of getting memory bank conflicts is low. But single-precision operands are 64 bits and the memory banks contain 32-bit halfwords. Here the access of a single operand involves two successive memory banks, still the number of memory banks is large enough. So we only expect effects that depend on the alignment of operand and result vectors.

The results are shown in Figs. 1 and 2. The figures show a grid of squares where the gray-scale of a square shows relative performance. White squares indicate best performance, black squares indicate worst performance. We show the results only for II and JJ varying from 0 to 31; the remainder of the figure is very similar.

We immediately see the effects of bank conflicts. However, the difference between best performance and worst performance is less than 5%. This is due to the microcode-controlled circuitry within the Cyber 205 that tries to avoid memory bank conflicts in vector instructions. This circuitry will buffer the three data streams if necessary. Also due to this circuitry a timing

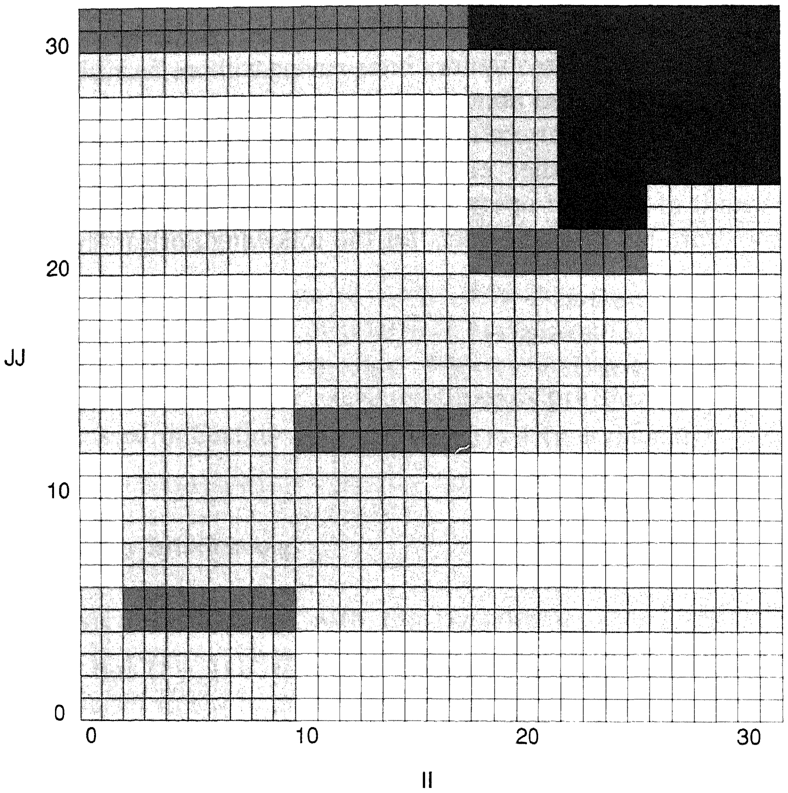

Fig. 1. CDC Cyber 205: addition.

difference will only be seen as a difference in startup time, so with longer vectors the relative difference would even be less.

2.2. *CDC Cyber 995* [4]

Although the CDC Cyber 995 has a cache, it is only used for scalar operations. So this machine is described in this section rather than in Section 3. Vector operations are from memory to memory (as on the 205). The number of memory banks is 32. Memory cycle time is 4 or 6 times processor cycle time, depending on the model. Timings may lead one to believe that the number of memory banks is larger, but this is because vector instructions probably fetch and store four vector elements at once. Although the manual does not say so, this would be consistent with the way cache works on these machines.

On this machine we performed the same tests as on the 205. The results are shown in Figs. 3 and 4. Not only is the result much less regular than on the 205; also the difference between best and worst performance is much bigger. The fastest case is about 30% faster than the slowest case (variation is from approximately 24 MFLOPs to approximately 35 MFLOPs). The 995 does not have circuitry to compensate for memory bank conflicts, so it is possible that a vector instruction will delay for a bank conflict every fourth cycle for each operand.

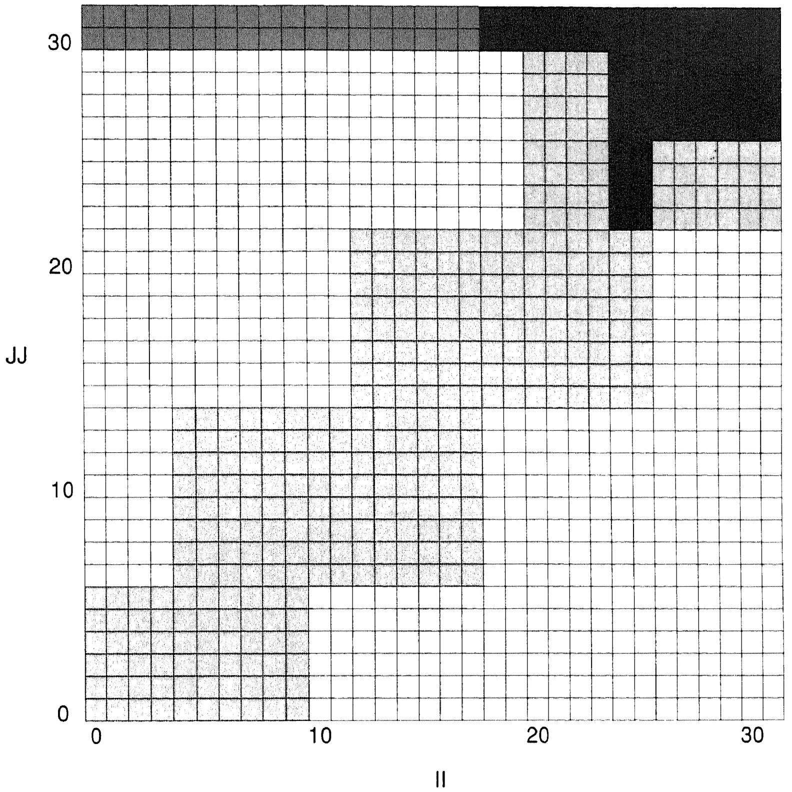

Fig. 2. CDC Cyber 205: multiplication.

Another interesting phenomenon here is that the picture is very asymmetric with respect to its input operands. In a few instances (II = 19, JJ = 0 amongst others) interchanging the operands would change the performance from near worst case to near best case. It is clear that it is a tedious job to obtain optimal performance on this machine. Luckily, if operands and results are a multiple of 32 apart, the performance is pretty good.

3. Machines with cache

As outlined in Section 1, when the processor does a memory request for reading, this request goes to the cache. The cache immediately returns the datum if it is available in the cache, otherwise it will read the datum from memory, store it in cache and forward it to the processor.

When a request for writing is performed the cache will store the datum and forward it to memory immediately (write-through). On some machines cache is organized such that the write to memory will be delayed until the space in the cache is needed for other data (copy-back). In many cases this will avoid successive memory writes to the same memory location.

It is clear that if a cache stores a datum, some other datum has to be removed. The algorithm by which the latter datum is chosen (the cache replacement algorithm) will influence the behaviour of the cache.

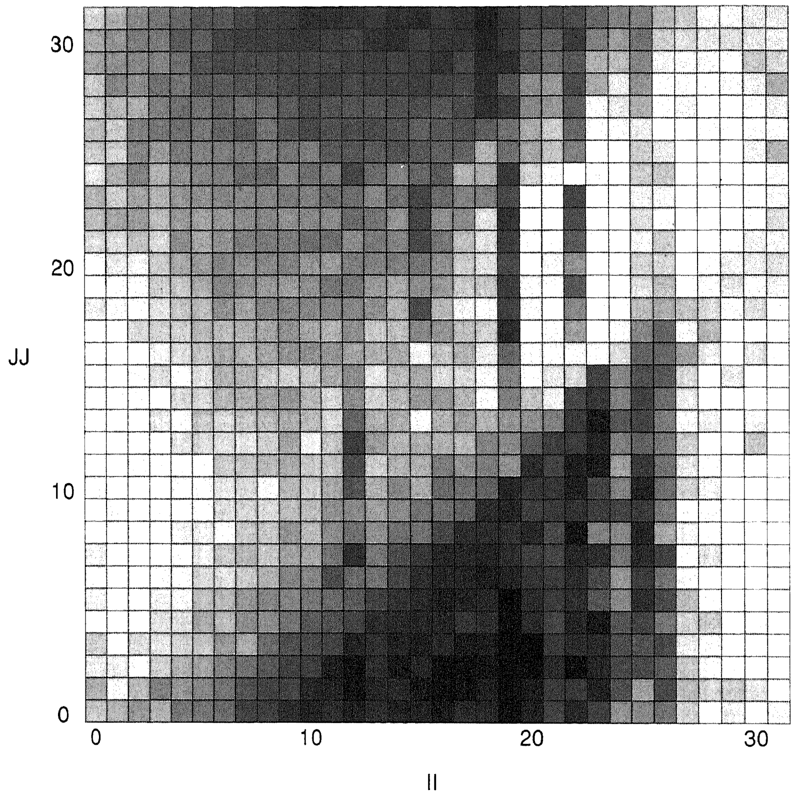

Fig. 3. CDC Cyber 995: addition.

The most important algorithms used are:

(a) FIFO. The cache is organized as a First In First Out queue. If a datum has to be stored, the oldest datum is removed. This algorithm is the simplest to implement.
(b) RANDOM. The datum that will be removed is chosen randomly.
(c) LRU (Least Recently Used). The datum that will be removed is the datum that has not been used the longest time.
(d) LFU (Least Frequently Used). In this case the frequency of use determines the choice.

There are numerous other algorithms in use, mostly designed to approximate LRU with a simpler implementation. Studies [2,7] that have been performed show that LRU and LFU are the best algorithms while FIFO is the worst. RANDOM is somewhere in between, but very close to LRU/LFU. Of course the relative merits of the cache replacement algorithms depend on the memory usage pattern involved. Also these studies apply to scalar code only and are not very relevant for vector processors.

Another factor that influences cache performance is the cache line size. In general when a datum is stored in cache this is not exactly the datum requested by the processor, but much more. The cache line size is the amount of data the cache will store each time. Cache line sizes vary from 4 to 512 bytes over different implementations. (Note: for the machines discussed here

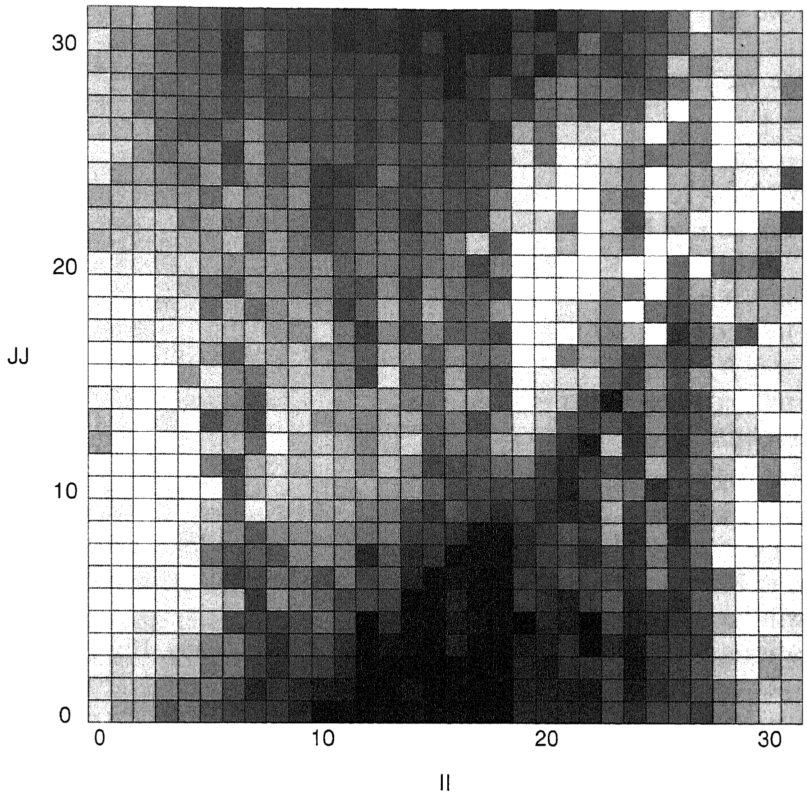

Fig. 4. CDC Cyber 995: multiplication.

single-precision floating-point is 32 bits, i.e., 4 bytes, so the line sizes vary from 1 to 1 single-precision floating-point numbers.)

Cache line size influences vector processing in an important way. Consider an operation on a vector with consecutive elements. When a request is performed for the first element a cache miss may be detected (i.e. the element is not in cache). The cache will load a cache line and return the element. Now when the second element needs to be accessed it is already in cache (at least if a cache line contains multiple elements) so processing is much faster. In this case the cache collaborates with the vector operation.

However, if the vector consists of nonconsecutive elements, when the second element has to be processed it is probably not in cache. In this case we might expect a cache miss on every element. In fact the cache fetches much more than is needed.

Caches may or may not be combined with memory interleaving. In general this will be invisible since every possible effect will be absorbed by the use of a cache line size of more than one element. E.g., if a cache line is large enough to contain an element from each memory bank we may just as well regard the memory as a one-bank memory with words just as large as a cache line. However, if a cache line cannot contain an element from each memory bank, we may expect memory bank conflicts for two cache lines. Among the processors studied in this report only one has that organization: the CDC Cyber 995, albeit only in scalar mode.

In this section we are mostly interested in the effects of cache misses on (vector) processor performance.

For this purpose we timed the following Fortran loop:

```
      II = 1
      JJ = 1
      DO 20 I = 1, IMAX
        DO 10 K = 1, 512
10        A(K,II) = A(K,II) + X*B(K,JJ)
        II = II + 1
        IF(II .GT. IIMAX) II = 1
        JJ = JJ + 1
        IF(JJ .GT. JJMAX) JJ = 1
20    CONTINUE
```

A and B are 512 by 1024 arrays.

Now let IIMAX = 1, and JJMAX ≠ 1. In this case the first source vector and the destination of the inner loop will not vary. The second operand of the inner loop will cycle through the first JJMAX columns of B. If JJMAX is small all B data can be kept in cache, but if JJMAX is large all B data references will be out of cache references.

Similar observations apply if JJMAX is fixed at 1 and IIMAX varies. But in this case we will also see the effect on writes that might be different from those on reads. Finally the timings were performed with IIMAX and JJMAX equal but varying.

The value of IMAX was such that initial cache misses would only slightly influence that total result.

We performed the timings on four different vector machines with cache.

3.1. Alliant FX/4 and Alliant FX/8 [1]

Cache size on the Alliant is 256 kbytes (FX/4) or 512 kbytes (FX/8). Cache line size is 32 bytes. The cache is copy-back, i.e. it does not transfer the result to memory immediately after computation.

The Alliants are multiprocessor machines. These machines allow different processors to operate concurrently on a single vector operation. The hardware allows two different forms of operation distribution:

(a) *Horizontal distribution*. In this case the first processor will do the first operation, the second processor the second operation, and so on in a cyclic way.

(b) *Vertical distribution*. Now the first processor will perform the first set of operations, the second processor the second set of operations, etc. The number of operations in a set is approximately equal for each processor.

For some reason the Fortran compiler generates horizontal distribution only, which is slower than vertical distribution in a number of cases. All the processors of the Alliant share the same cache, so we may expect different results for different numbers of processors. (This statement is not exactly true, but the actual implemented hardware is not detectable.)

In Figs. 5 and 6 the results are plotted for the FX/4 with 1 and 4 processors respectively. Figures 7, 8 and 9 show the results for the FX/8 with respectively 1, 4 and 8 processors. Note

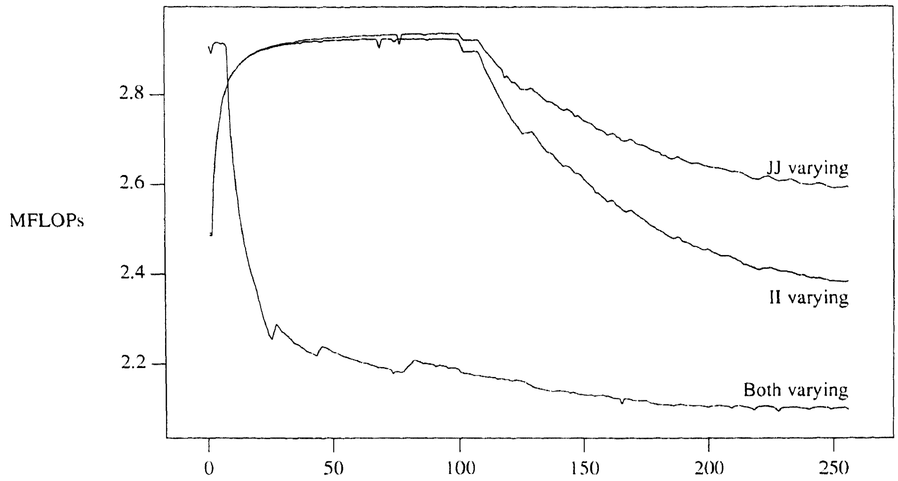

Fig. 5. Alliant FX/4: one processor.

that the horizontal scaling in the figures for the FX/8 is different from that of the FX/4 because of the difference in cache size.

One interesting point to note is that performance degradation is much less when only one processor is used. E.g. taking IIMAX = 1 and varying JJMAX we find on the FX/4 a degradation from 2.9 MFLOP to 2.6 MFLOP when only one processor is used. When four processors are used degradation is from 9.9 MFLOP to 7 MFLOP.

Also we easily see that performance is best if only one of the sources varies, and performance is worst if both sources and the destination vary. In the latter case we even see a nearly instantaneous performance hit that is not warranted by out of cache references. The reasons for this are not clear. Another unexplained feature is the rise of the curve for low values of IIMAX and JJMAX.

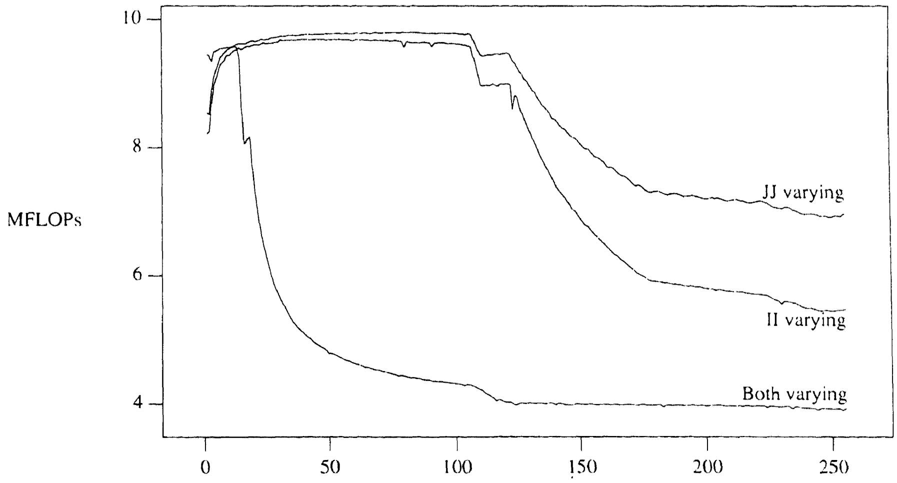

Fig. 6. Alliant FX/4: four processors.

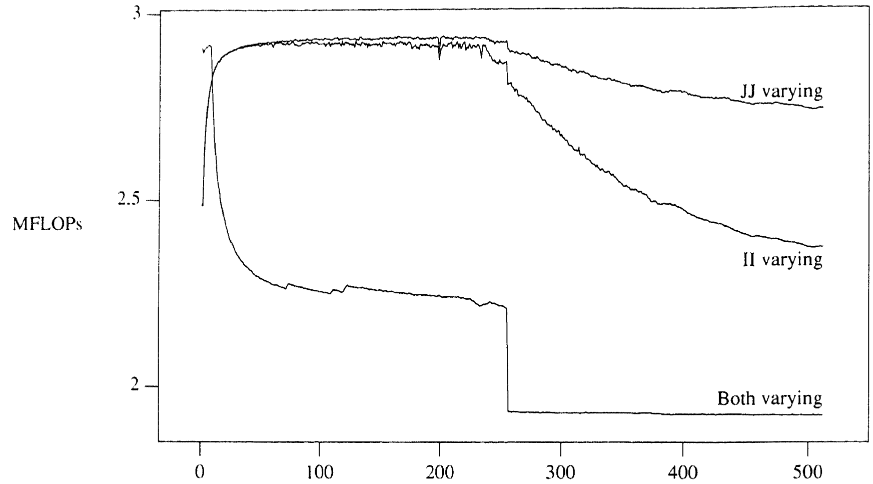

Fig. 7. Alliant FX/8: one processor.

Looking at the pictures we also see that degradation is not immediate if the cache limit is hit, but that degradation is gradual. This is consistent with a RANDOM cache replacement algorithm (the longer we postpone reuse of a datum, the less likely it will be in cache). This behaviour is not consistent with LRU and LFU algorithms: for these we expect an immediate degradation if the cache limit is hit, as the program cycles through the data.

Comparing the FX/4 with the FX/8 we see that increasing the cache size does help also if the total amount of data is larger than the cache, since degradation is much slower.

The figures for the FX/8 are a bit more ragged because that machine was not completely unloaded when the timings were done.

However, in reality the cache does not use a RANDOM replacement algorithm. Although the actual cache strategy is not published, communication with Alliant Computer Systems Corporation taught us that each datum in memory has a fixed place in the cache (direct

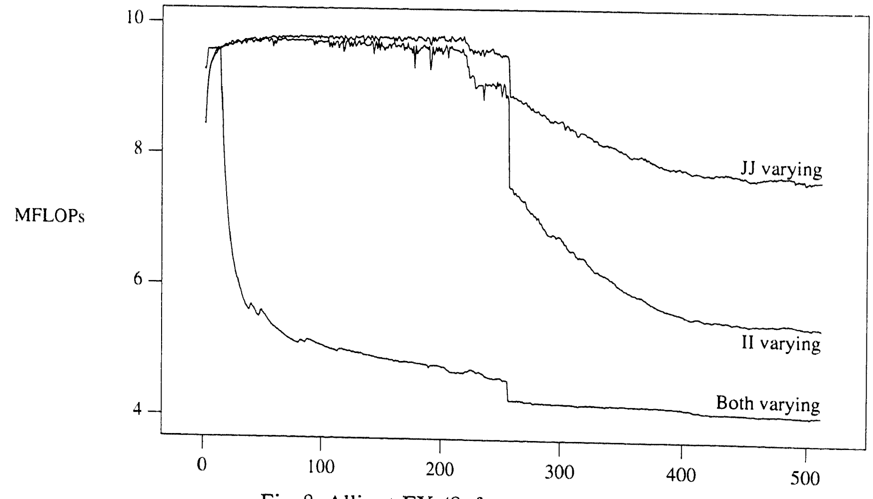

Fig. 8. Alliant FX/8: four processors.

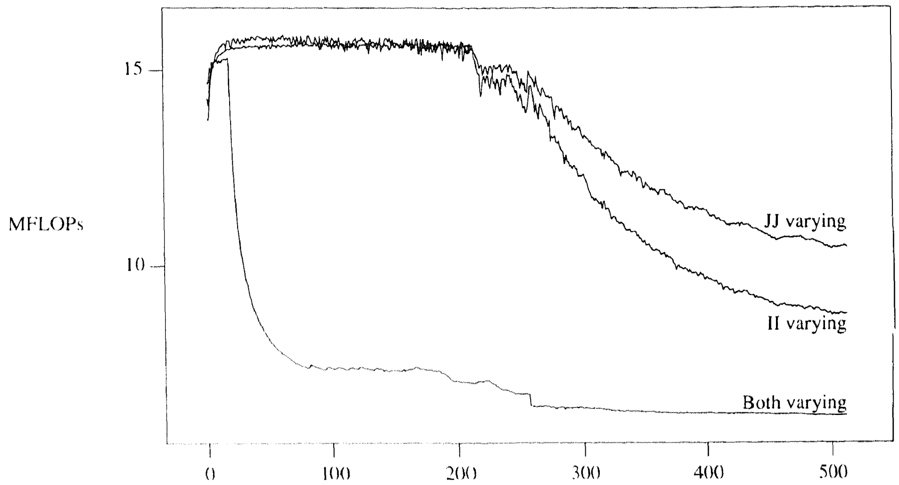

Fig. 9. Alliant FX/8: eight processors.

mapped cache). So a datum is removed if a newer required datum occupies the same place in cache. Indeed, in general this makes the cache look like a cache with RANDOM replacement algorithm, but it is possible to construct examples where cache performance is particularly bad. One such example can be found when one steps through an array with steps that are large powers of 2. We will have a short look at this phenomenon below.

These machines also allow for operation on noncontiguous vectors. As indicated before we may expect performance worse than for contiguous vectors, because the cache will load more than is needed. Indeed, when we do similar timings using rows rather than columns we see that performance is much worse. Table 1 shows the results (in MFLOP). In this case, if IIMAX is 1, array A is contiguous; if JJMAX is 1, array B is contiguous. These results are approximately constant for all values of IIMAX, c.q. JJMAX. If the results is not contiguous (IIMAX not equal to one), the performance is clearly worse than if the result is contiguous. Also the influence of the number of processors is much smaller than when columns are used. Cache size has some influence, but that is not significant. Here clearly the actual cache strategy used has influence, although not all these effects can be explained by this. However, as this is the only machine studied that does allow noncontiguous vectors, we did not pursue this further.

3.2. *Gould NP1* [5]

Cache size on the Gould NP1 is 16 kbytes. (In fact it is 32 kbytes but half of that is used as instruction cache.) When we look at the pictures (in Fig. 10) we see an immediate performance

Table 1

	# processors	JJMAX varies	IIMAX varies	both vary
FX/4	1	1.6	0.9	0.7
	2	2.8	1.1	0.9
FX/8	1	1.6	1.0	0.8
	4	2.8	1.2	0.9
	8	3.1	1.3	1.0

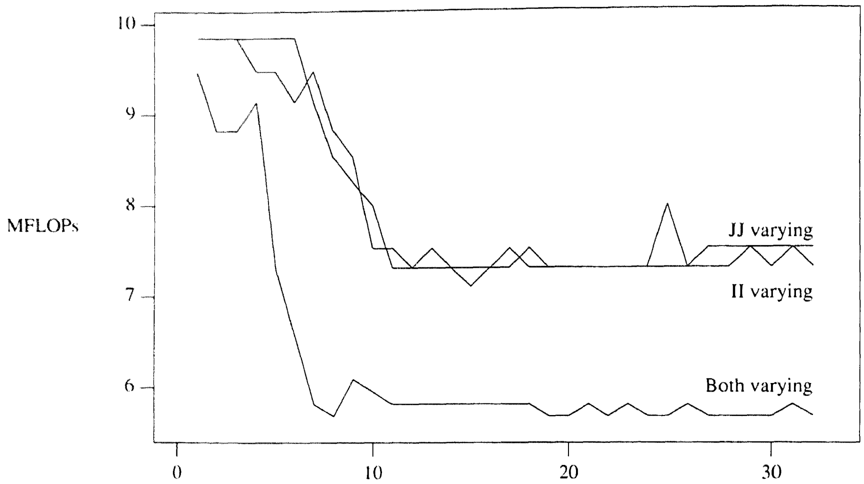

Fig. 10. Gould NP1.

degradation if the problem size exceeds the cache size. This is consistent with either FIFO or LRU as cache replacement algorithm. Also we see that performance degradation is much more severe if both operands and the result are out of cache. On the other hand, there is not much difference if we vary both the destination and one of the sources (varying II) or only one of the sources (varying JJ). This might indicate a delayed write where data is written to memory only if there is no memory contention.

The occasional peaks in the figures are due to the poor resolution of the timing clock.

3.3. *IBM 3090-VF* [6]

The cache on the IBM 3090-VF is 64 kbytes. The cache is organized in four equally sized ›anks of 16 kbytes. Each datum from memory can be placed in exactly one place in any of the

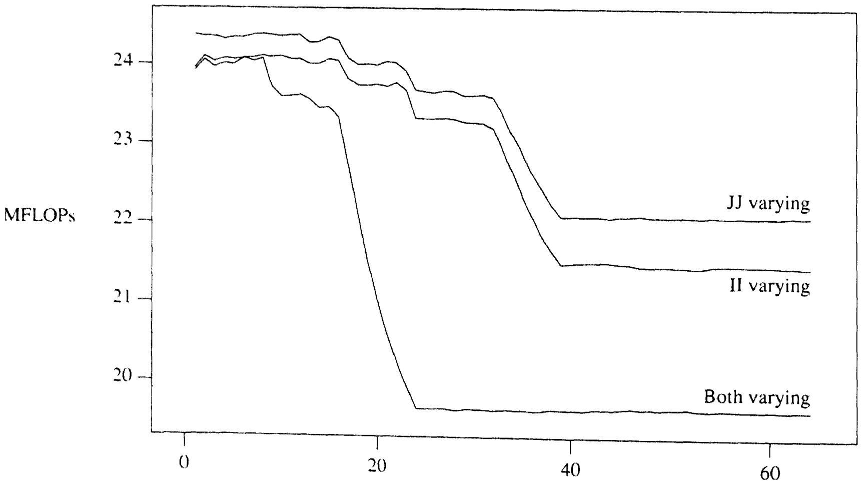

Fig. 11. IBM 3090-VF.

four cache banks (four-way set associative cache). The cache replacement algorithm for the individual sets is FIFO.

Figure 11 displays the results for the IBM. We can easily see the results from the cache organization in this figure; there are a number of distinctive performance hits.

Overall degradation in performance is small (from about 24 to about 22 MFLOP).

4. Vector registers

Many current vector processors have vector registers. These may or may not be combined with a cache memory (e.g., Cray and NEC machines have vector registers but no cache). Vector registers serve very well if the data can be kept within the registers for many operations, because all effects of slow memory will be spread out over the operations. Memory effects will be most visible if only one operation can be performed on the vector registers. In that case a typical operation requires two loads and a single store of a vector register.

If there is a cache, performance degradation is mostly due to the effects of the cache, and this has been covered in the previous section.

If there is no cache and only a single path from the vector registers to memory the only problem we will see is if the data is not contiguous (except of course in a shared memory multiprocessor machine; but that is beyond the scope of this article). For contiguous data on an interleaved memory system conflicts will not arise.

However, if there are multiple data paths from the processor to memory we may see some effect of this, but this will not very much influence performance. The reason is that if a bank conflict exists when an operation starts, that operation will be delayed until the bank conflict is solved. So all effects will only be seen in start up times, just as with the CDC Cyber 205. For this reason no extensive timings for such machines have been carried out.

5. Conclusions

Because of the variety of situations no general conclusions can be drawn. It is only possible to draw some conclusions for particular strategies.

For memory to memory vector machines the strategy of the CDC Cyber 205 (where delays are built into cover-up bank conflicts) performs quite well. Performance of such a machine without this cover-up would be a bit erratic, and very difficult to control by the programmer.

Cache memories perform fairly well, but the cache must be large enough. The Gould NP1 suffers from too small a cache to perform well on large problems. The IBM 3090-VF has a reasonably well designed and large enough cache (although the FIFO algorithm might give problems) but evidently the cache is not much needed for vector operations; perhaps memory to memory operations would be better for this machine. The cache of the Alliant appears to be well-designed and large enough to maintain reasonable performance for quite large problems. However, the use of noncontiguous vectors should be avoided on these machines.

References

[1] FX/Series, Architecture Manual, Publication No. 300-00001-C, Alliant Computer Systems Corporation, Littleton, MA (1988).

[2] L.A. Belady, A study of replacement algorithms for a virtual storage computer, *IBM Systems J.* 5 (2) (1966) 78–101.

[3] CDC Cyber 200 Model 205 Computer System Hardware Reference Manual, Publication No. 60256020 rev B, Control Data Corporation, St. Paul, MN (1982).

[4] CDC Cyber 180 Computer System Model 990, CDC Cyber 990E and 995E Computer System, Virtual State, System Description and Functional Descriptions, Hardware Reference Manual Vol. 1, Publication No. 60462090 rev D, Control Data Corporation, St. Paul, MN (1987).

[5] NP1 Central Processing Unit (CPU) Model 4020 Reference Manual, Publication No. 301-006600-000, Gould Inc., Fort Lauderdale, FL (1987).

[6] *IBM Systems J.* 25 (1) (1986) Special Issue on to the 3090.

[7] A.J. Smith, Cache memories, *Comput. Surv.* 14 (3) (1982) 473–530.