Memory Interleaving

Norman Matloff
Department of Computer Science
University of California at Davis

November 5, 2003

1 Introduction

With large memories, many memory chips must be assembled together to make one memory system. One issue to be addressed in *interleaving*.

First, some assumptions. Let us assume throughout this tutorial that our machine is not byte-addressable, i.e. individual bytes do not have separate addresses.[1] Also, assume that a bus cycle is the same as a CPU clock cycle, and that the the memory is broken up into m physically-separate components called *modules*, with m chosen so that the access time of a module is m bus cycles.[2]

To illustrate the role of interleaving, suppose we wish to set up a memory system of 256M words, consisting of four modules of 64M words each. Denote our memory modules by M0, M1, M2 and M3. Since 256M is equal to 2^{28}, our system bus would have address lines A0-A27. Since 64M is equal to 2^{26}, each memory module would have address pins A0-A25. The question is which bus lines to connect to which pins. Clearly, we need to devote two of the bus lines to determining which of the four modules a requested word is in, since $4 = 2^2$, but which two lines should we choose?

1.1 High-Order Interleaving

Arguably the most "natural" arrangement would be to use bus lines A26-A27 as the module determiner. In other words, we would feed these two lines into a 2-to-4 decoder, the outputs of which would be connected to the Chip Select pins of the four memory modules. If we were to do this, the physical placement of our system addresses would be as follows:

[1] If our machine has 32-bit word size and is byte-addressable, as most machines are today, we would devote two bits of the address to select byte within word, and the rest of this tutorial would apply to the remaining address bits.

[2] In the low-order interleaved case, a module is sometimes called a **bank**.

address	module
0-64M	0
64M-128M	1
128M-192M	2
192M-256M	3

Note that this means consecutive addresses are stored within the same module, except at the boundary. The above arrangement is called *high-order interleaving*, because it uses the high-order, i.e. most significant, bits of the address to determine which module the word is stored in.

1.2 Low-Order Interleaving

An alternative would be to use the low bits for that purpose. In our example here, for instance, this would entail feeding bus lines A0-A1 into the decoder, with bus lines A2-A27 being tied to the address pins of the memory modules. This would mean the following storage pattern:

address	module
0	0
1	1
2	2
3	3
4	0
5	1
6	2
7	3
8	0
9	1
etc.	etc.

In other words, consecutive addresses are stored in consecutive modules, with the understanding that this is mod 4, i.e. we wrap back to M0 after M3.

2 Comparison

2.1 Some Sample Applications

It is important to keep some actual applications in mind, such as:

- Gaussian elimination. Recall that a typical operation is to add a multiple of one row of our matrix to another row of the matrix.
- Matrix multiplication. Recall that to do AB = C for matrices A, B and C, we get C_{ij} by taking the inner product ("dot product") of A's i^{th} row and B's j^{th} column.

- Image processing, such as in Adobe Photoshop or the open-source GIMP. Here element (i,j) element in our matrix would be the brightness at the pixel (i,j).[3] If we wish our image to look smoother, then at each pixel we would average the brightness at neighboring points, and make this value average our new brightness at the given point.

2.2 The Notion of Stride and Application to Vector Machines

The C/C++ standard specifies that two-dimensional arrays must be stored in *row-major order*, i.e. elements in the same row have consecutive memory addresses. So, whenever we access a row of a two-dimensional array, as we do in the Gaussian elimination example above, the sequence of memory addresses generated by our program go up in increments of 1. We say that the *stride* is 1.

On the other hand, consider the matrix multiplication example above, and suppose each of the matrices is nxn. Then in the case of matrix B, we are accessing a column, which means that the sequence of memory addresses generated by our program go up in increments of n, which we call a stride of n.

The point is that if we have a stride of 1, low-order interleaving works quite well. We can do m memory accesses in parallel, because the accesses are to different modules. Low-order interleaving gives us no advantage if the stride is m, though; in that case, all the words we are accessing are in the same module, so no parallel operation can be done. So for instance in the matrix multiplication example, if n is equal to (or even is a multiple of) m, the access of the columns in B will be slow.

Note that if the stride is equal to m/2, we will keep two modules busy at once, representing a speedup but not a very good one. One can prove that we attain full speedup if and only if m and the stride s are relatively prime.[4]

Low-order interleaving is vital to the construction of *vector machines*, such as the Cray and the Fujitsu VP/2000, which are capable of doing mathematical operations on entire arrays. In such a machine, in addition to having a scalar addition instruction, e.g.

```
ADD A,B
```

there would also be a vector version, e.g.

```
VADD X,Y,S
```

where **X** and **Y** are arrays and **S** is the stride. If fast access to memory were not available, these machines simply would not be able to attain much speed.

Low-order interleaving also is useful in systems which have cache memories. Since a block replacement[5] has a stride of 1, low-order interleaving allows us to do the replacement very quickly.

[3] We might have three matrices, for each of the three primary colors.

[4] This comes from group theory in modern algebra.

[5] Or write back, if we have a write-back policy.

2.3 The Role of Interleaving in Shared-Memory Multiprocessors

High-order interleaving is useful in shared-memory multiprocessor systems. Here the goal is to minimize the number of times two or more processors need to use the same module at the same time, a situation which causes delay while processors wait for each other. If the system is configured for high-order interleaving, we can write our application software in a way to minimize such conflicts.

In matrix applications, for instance, we can partition the matrix into blocks, and have different processors work on different blocks. In image processing applications, we can have different processors work on different parts of the image. Such partitioning almost never works perfectly—e.g. computation for one part of an image may need information from another part—but if we are careful we can get good results.

3 Refinements to Low-Order Interleaving

3.1 Address Skewing

One refinement to low-order interleaving involves *skewing* of the distribution of addresses to the memory modules. Let's again use the case of four modules as an example, with the following address assignment pattern:

address	module
0	0
1	1
2	2
3	3
4	1
5	2
6	3
7	0
8	2
9	3
10	0
11	1
12	3
13	0
14	1
15	2
etc.	etc.

Compare this to the table in Section 1.2. There, for each group of four addresses whose first address is a multiple of 4, the module numbers were 0, 1, 2 and 3, in that order:

address group	module order
0,1,2,3	0,1,2,3
4,5,6,7	0,1,2,3
8,9,10,11	0,1,2,3
12,13,14,15	0,1,2,3
etc.	etc.

But in the skewed arrangement, each such group is shifted rightward from the previous one:

address group	module order
0,1,2,3	0,1,2,3
4,5,6,7	1,2,3,0
8,9,10,11	2,3,0,1
12,13,14,15	3,0,1,2
etc.	etc.

In general, instead of address i being stored in module i mod m, that address would be stored in module $(i + \lfloor i/m \rfloor) \, mod \, m$.

You can readily see that with skewed interleaving, we see get full speedup when doing matrix row operations, since we can keep m modules busy at once.[6] But in addition, we also get full speedup for column access as well.

Note, though, that there are still problems with other types of access, such as cases in which we need to access all diagonal elements of a matrix.

3.2 Prime Number of Modules

Typically interleaving schemes take m to be a power of 2, for ease of implementation. For low-order interleaving, for example, one determines module number by merely using the $log_2 m$ least significant bits of the word address.

However, as we noted earlier, we get full speedup if the stride is relatively prime to the number of modules. This implies that the best value of m is prime! The hardware is more difficult to implement for a mod-m operation when m is not a power of 2, but with modern fast division algorithms it is feasible.

3.3 Superinterleaving

Up to this point, we have assumed that the number of memory modules is equal to the access time, in bus cycles, of one module. The concept of *superinterleaving* relaxes that assumption.

For instance, in our examples above with m = 4, suppose we set m = 8 but the module access time is still 4

[6] This is not quite true if our access does not start and end at boundaries of the address groups. For example, in the table above, suppose we need to access addresses 3 through 13. We can access addresses 4-11 at full speed, but conflicts would occur for address 3, 12 and 13.

bus cycles. Because of the latter constraint, we cannot ever achieve a speed of any greater than one memory access per cycle. However, we can reduce the number of conflicts.

For example, with a stride of 4, we could not get any speedup at all in a system for which m = 4, since all requested words would be in the same module. But with m = 8, we have fewer such clashes, and thus can keep two modules busy at the same time, doubling effective access speed.

3.4 Software Solutions

Recall that in our matrix multiplication example, we were doing column access to the matrix B, which was slow, due to complete module conflict. One possible solution to this would be to write our software so that it stores B^t, the transpose of B, instead of B. The column access of B would then be row access of B^t, which would be done quickly.

Another possibility would be to store B with a fake extra column. This would increase the stride within a column by 1, eliminating the conflict problem.

Note that none of these solutions is perfect. Storing B^t instead of B, for instance, may cause problems with other computations involving B in the same program.