4. Memory

Looking at the instruction set of a typical machine you might conclude that the only memory consists of registers and main memory. But in fact, main memory is surrounded by other levels of memory a higher one to speed it up and a lower one to increase its capacity. Both are invisible to the casual user; but both have important consequences for scientific programming.

For the CPU to perform efficiently it must be able to communicate with main memory at rate that approximates its clock cycle. Unfortunately, fast memory is expensive, and main memory in all but some costly supercomputers is too slow to feed the CPU. The cure is to have a **cache** of fast memory as acting as a buffer between main memory and the CPU. As long as memory references are to items in the cache, memory transaction are completed swiftly enough to keep the CPU humming. Of course, when a reference is made to an item not in cache, things slow down while the cache is updated. This means that the scientific programmer has a strong interest in coding in such a way that the items he or she needs are usually in the cache.

Many algorithms for scientific applications — computational fluid dynamics for one — end up manipulating huge amounts of data, far too much to be contained in main memory. Hence the data must be placed in a backing store — usually a disk — and brought into main memory as needed. One way to do this is to write programs that explicitly move the data between the disk and memory. In some applications this is exactly what should be done. However, most machines support a **virtual memory** that allows the programmer to code as if a very large main-memory were available. The combination of hardware and software that supports virtual memory moves things in out of the actual main memory automatically without programmer intervention.

Thus we see that memory is organized in a **memory hierarchy.** At the top are the registers that are the working memory of the CPU. Next comes cache memory, then main memory, and finally at the bottom virtual memory. As we move from top to bottom the memories become slower but at the same time larger.

We will begin our treatment of the memory hierarchy in the middle with main memory. We will then go on to treat cache memory and finally virtual memory. The section concludes with some hints on how to program to use the hierarchy efficiently.

4.1. Main memory

Synopsis: Main memory consists of random access memory (RAM). There are two types. SRAM is fast and expensive. DRAM is slower and cheaper, and it the customary choice for main memory. Main memory communicates through the memory bus with the help of a memory controller. The memory cycle is the time required to complete a memory transaction. By establishing independent memory banks, memory references can be pipelined so that the effective memory speed is the memory cycle divided by the number of banks. However, an unfortunate choice of stride can slow down this process.

—

4.1.1. Generalities

Main memory is an invariably a **random access memory** (RAM). The term 'random' refers to the fact that you can take an address at random and access the corresponding memory location a constant amount of time, which we will call a **memory cycle.** RAM's come in two flavors: static RAM's or **SRAM**'s and dynamic RAM's or **DRAM**'s.

SRAM's are fast, expensive, and stable. By stable we mean that once a item is written into memory it stays there — at least until the power is turned off. Because cycle times for SRAM's are in range of a few nanoseconds, SRAM's are used in caches or sometimes in the main memory of supercomputers.

DRAM's are cheap and capacious because their circuitry is simple. For the same reason, they are comparatively slow — over ten times slower than SRAMS. Moreover, that anything written to a DRAM will fade away in a few milliseconds unless it is refreshed. But because they offer large, affordable memories, most main memories made of DRAM chips.

Main memory has at least one **memory controller.** This is a piece of hardware between the memory bus and the memory itself that is manages memory addresses and data to be loaded or stored. The controller is also responsible for refreshing the DRAM.

4.1.2. Interleaved memory

Although most main memories are addressable by bytes what is actually transferred is an aligned word typically of 32 or 64 bytes. One way to speed up main memory is to lengthen the word. However, this option must be built into the fabric of the computer when it is designed.

A more flexible way to speed up main memory is to pipeline memory accesses. Specifically, memory is divided into independent banks. The figure below shows the beginning of a small memory divided into four banks.

Bank 1	Bank 2	Bank 3	Bank 4
00000000	00000001	00000010	000000011
00000100	00000101	00000110	000000111
00001000	00001001	00001010	000010111
00001100	00001101	00001110	000011111
00010000	00010001	00010010	000100111
...	...	...	...

The address space (here given in units of words, not bytes) is wrapped horizontally around the banks (we say that it is **interleaved** among the banks). Let us suppose that the addresses are fed to the memory in increments of one at the rate of four address per memory cycle. The following table shows how the pipeline operates over time.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	1 5
Bank 1	r	-	-	d	r	-	-	d	r	-	-	d	r	-	-
Bank 2	-	r	-	-	d	r	-	-	d	r	-	-	d	r	-
Bank 3	-	-	r	-	-	d	r	-	-	d	r	-	-	d	r
Bank 4	-	-	-	r	-	-	d	r	-	-	d	r	-	-	d

Data from the first bank is requested ('r' in the diagram) at time 1, and the data is delivered (d) at time 4. Thus at time 5, the first bank is ready to honor another request for data. Data from the second bank is requested at time 2; the data is delivered at time 5, and new data is requested at time 6. And so on. The effect of all this is that after a startup time of four memory cycles, memory references are processed at a pipelined rate of four per cycle.

Interleaved memories require sophisticated memory controllers. One reason is that if the individual addresses are passed through the memory bus they will compete with the flow of data and slow the effective rate of transfer. A solution is to send the controller the starting address and the number of words to be transferred. The controller can then compute the address for the individual memory banks on the fly, leaving the memory bus free to transmit data. An alternative is to have a separate data bus. In this case the CPU can send addresses over the memory bus without interfering with data transfer.

Recall (§3.5.6) that a memory stride is the distance separating successive memory references. Interleaved memories function well when the transfers have a stride of one word. Other strides may give trouble. For example, consider passing down a column of the array `A` in Figure 3.5. The memory references have a stride of four double words. Hence the elements of a column all lie in the same bank of our four bank memory, and there will be no pipelining.

Note that in this example the stride is just the trailing dimension in the declaration

```
double A[5][4]
```

that defines `A`. Consequently, one way of avoiding the this problem in `C` is to choose the trailing dimension so that the references have to cycle through all the banks before it repeats a bank. It turns out that a necessary and sufficient condition for this is that the stride and the number of banks have no common divisor (see Exercise 4.3). Since the number of banks is always a power of two, a trailing dimension that is odd will always satisfy this condition. (In Fortran, which stores its arrays in column-major order, it is the leading dimension that determines the stride.)

It is important to keep these observations in perspective. In most machines — from PC's to high-end workstations — the CPU does not reference main memory directly but gets its data from an intermediate buffer called a cache. When the cache is refreshed, as it must be periodically, it is done by reading blocks of main memory with a stride of one. In this case, changing the stride in an array may change the performance of the cache, but a banked main memory will continue to tick along at an optimal rate.

4.2. Cache memory

Synopsis: Cache memory is a fast buffer between main memory and the CPU. Transfer of data between the two takes place in blocks of uniform size. When a memory reference is made the machine first looks for the block in cache. If it is there (a cache hit) the reference is honored from cache. If not (a cache miss) the block is moved in from main memory, perhaps displacing another block in cache.

There are three strategies for mapping blocks from memory to blocks in the smaller cache. The fully associative mapping lets a memory block end up anywhere in cache — flexible, but too costly. Direct mapping assigns a unique cache block to any memory block. Set associative mapping is a compromise between the two, leaning in practice toward direct mapping.

Read hits and misses are easier to handle that write hits and misses, since they do not change memory. There are two ways of treating write hits: write-through and write-back. Write-through writes both the cache and main memory. The slow speed of main memory can be a bottleneck here. A stopgap cure is to provide a buffer to hold pending write-throughs. Write-back writes only to the cache, and waits until a block is about to be overwritten to write it back to main memory.

Most machines have two caches. A superfast level-1 cache on the CPU chip and a level-2 off-chip cache between it and main memory. In addition, there is usually an instruction cache to buffer instructions coming in from main memory.

Because peripheral devices can read an write main memory independently of the CPU, maintaining coherency between cache and main memory is a challenging problem.

—

4.2.1. Introduction and nomenclature

A **cache memory** is a fast memory that buffers pieces of main memory so that they can be quickly read by the CPU. Its address space is divided into **blocks** of fixed number of bytes. The block size is always a power of two as is the number of blocks. The address space of main memory is also divided into blocks of the same size. Since there are more blocks in main memory than in the cache, not all main-memory blocks can reside simultaneously in cache. Consequently, blocks are moved back and forth between main memory and cache. When the CPU makes a memory reference, it looks in cache for the block corresponding to the memory reference. If it is there — an event called a **cache**

hit — it processes the memory reference from cache. If not — a **cache miss** — it brings in the block from main memory to cache before processing the memory reference.

Updating the cache when a miss occurs requires time — time that is called the **miss penalty.** To counter the miss penalty, the new cache block must be used enough times so that the savings in time are greater than the miss penalty. Fortunately, many programs have the property that if they reference a memory location then they are likely to make a reference to a nearby location in the near future. Such programs are said to have **locality of reference.** Later, we will discuss programming techniques for improving locality of reference.

The word 'cache' comes from the French *cacher* meaning to hide. The American Heritage Dictionary defines it as "a hiding place used especially for storing provisions." A hiding place it certainly is, at least in the sense that its contents and workings are invisible to the programmer. For a memory cache the 'provisions' are the contents of memory. But keep in mind that the word has spread throughout computer science, where the provisions vary. Operating systems have file caches. Your browser has web caches that store previously downloaded files and html pages. The point is that the unqualified use of the word 'cache' can cause confusion unless the context is clear. When in doubt, qualify.

4.2.2. Seven questions

Turning now to the details of memory cache, we will consider the following seven questions, whose answers will give a reasonably clear picture of how a memory cache works.

1. Where in the cache are memory blocks constrained to lie?
2. How does one determine if a memory block is in the cache?
3. When a new block is brought in, what block does it displace?
4. What happens on a read hit?
5. What happens on a write hit?
6. What happens on a read miss?
7. What happens on a write miss?

4.2.3. Cache organization

We will answer the first two questions together. For purposes of illustration we will assume that main memory has a 32 bit address space and that the cache consists of 2^{10} blocks of 32 bytes.

Regarding location and recognition of blocks, the ideal would be if a block in main memory could map to any block in cache. This arrangement is called a **fully associa-**

tive cache. It gives us maximal flexibility in choosing which blocks to replace. But consider the problem of determining if a block is in the cache.

Let's write the address generically of a memory location in the form

$$\underbrace{\texttt{ttttttttttttttttttttttttttt}}_{27}\underbrace{\texttt{ooooo}}_{5} \qquad (4.1)$$

The address of the block in main memory is

```
ttttttttttttttttttttttttttt00000
```

and the **offset** `ooooo` is the amount that must be added to the beginning of the block to get the byte being referenced. It follows that the block is uniquely determined by the 27 bit **tag field**

```
ttttttttttttttttttttttttttt
```

This suggests that each block of cache be given an extra piece of memory to store the tag field of the block it currently contains. When a memory reference is made, the blocks are searched for one having the appropriate flag. If one is found, we have a cache hit; if not, a miss.

The difficulty with this is that we have to search the flags of $2^{10} = 1{,}024$ blocks. This search must be performed in a CPU cycle or two — the speed at which the cache is supposed to complete memory transactions. Although it is possible to design a logic circuit that will do this, it will have an impractically large number of gates. But if we restrict the mapping of memory blocks to cache blocks we can simplify the search.

The common procedure is to divide the cache into sets of, say, four consecutive blocks. A memory block is mapped to set, but within that set it can occupy any block. This gives some flexibility, but to determine if the memory block is in cache only four cache blocks must be searched. Such an arrangement is called a **4-way set-associative cache.**

It's instructive to see how all this translates into bits. Let's write the address of a byte in the form

$$\underbrace{\texttt{ttttttttttttttttttt}}_{19}\underbrace{\texttt{ssssssss}}_{8}\underbrace{\texttt{ooooo}}_{5} \qquad (4.2)$$

The **set field** `ssssssss` has eight bits, just right to index 2^8 four block sets, which exactly fits into our sample cache. The cache addresses of first bytes of the blocks in the set are

```
ssssssss0000000
ssssssss0100000
ssssssss1000000
ssssssss1100000
```

Because the set field `sssssss` uniquely determines the set, we can use the 19 bit field

```
ttttttttttttttttttt
```

as a tag to determine the presence or absence of the block in cache, as described above. However, in this case we only have to search four blocks for a tag match.

We can increase or decrease the associativity by decreasing or increasing the size of the set field. For example, if we reduce the number of `s`'s in (4.2), replacing them on the right by `t`'s, we decrease the number of sets and increase the number of blocks per set — and hence increase the associativity of the cache. As an extreme example, when all the `s`'s are replaced by `t`'s we are back to a fully associative cache [see (4.1)]. On the other hand, if we increase the number of `s`'s to ten, there is only one block per set. This means that each memory block is associated with a unique cache block. This arrangement is called a **direct mapped cache.**

In general, some associativity helps cache performance. But the law of diminishing returns quickly sets in, and most memory caches are direct mapped or are 2-way or 4-way set associative.

4.2.4. Replacement strategies

Turning now to our third question, when a block is brought into cache, some block must be displaced. For direct mapped caches, there is no choice, since the incoming block can only map to one cache block, which must be displaced. With set-associative caches the incoming block can displace any block in the set, and the problem is which block to choose.

One effective strategy displaces the **least recently used (LRU)** block. Specifically, blocks in the set are ordered according the last time they received a hit. At the bottom of this ordering is the LRU block. The rationale for this choice is that since the LRU block has been unused longest it is likely to remain unused. It requires some additional hardware to update the ordering after a hit or miss to the set. But for a cache of low set associativity this is not a great burden.

A popular, easily implemented alternative is to choose the block to displace at random. In practice this works almost as well as LRU. A drawback is that if the choice is truly random, then the behavior of a program will not be not reproducible from run to run.

4.2.5. Response to read and write hits

Our forth question is what to do about a read hit. This answer is rather simple. Complete the memory transaction from cache and continue. The reason for the simplicity is that a read changes neither the cache or the main memory.

The fifth question — what to do about a write hit — is more difficult. The problem is that anything written to cache must ultimately be reflected in main memory. There are two alternatives that go under the names of write-back and write-through.

When **write-back cache** gets a hit, it writes the cache but not main memory. It also sets a flag, called a **dirty bit**, that says that the block in question has been written. In the short run this is fast, but when the block is displaced it must be written back in its entirety to main memory (if its dirty bit is set) before the new block can be transferred to cache.

When a block in a **write-through cache** is hit, the cache is written but main memory is also written. This ensures the identity of the cache and main memory, so that there is never a need to write back a displaced block. On the other hand, writes to main memory are slow and there is a danger that they will degrade the machines performance. The problem can be partially alleviated by providing a buffer to hold the writes until they can be processed. But such a buffer must be finite; and once it has been filled, things will slow down. This could easily happen with the AXPY program in Figure 3.2.

4.2.6. Response to read and write misses

The sixth question concerns what to do with a read miss. Again the answer is simple. Swap in the memory block that was not in the cache. On the principle of locality, we hope for another reference to the block in the near future, a reference which can then be honored from cache.

The answer to the last question — what to do with a write miss — depends on whether the cache is write-back or write-through. If the cache is write-back, it is natural to swap in the missed block, so further writes can be honored in cache. If the cache is write-through, it is not clear that anything is gained by swapping the block in. Hence doing nothing is a plausible strategy.

4.2.7. Multiple caches

Most machines have a small, very fast cache on their CPU chip (often called the **level 1 cache** or **L1 cache**). Because it is small, we can expect a high miss rate. And because it is fast, updating at the rate of main memory will slow it down. The cure for this problem is to place an L2 cache between main memory and the L1 cache. This cache need not reside on the chip, but it must have a fast dedicated connection with the CPU. Some machines even have a L3 cache between the L2 cache and main memory. Although this kind of cache hierarchy complicates the hardware of the machine, it can speed up memory transactions at the CPU, where speed is most needed.

A second kind of additional cache is the **instruction cache.** We have seen that the execution of each instruction in a program begins begins by fetching the instruction from memory. Since this is an ordinary memory reference, it can be cached like any other. But then it must compete with load-store instructions for access to the cache, resulting in structural hazards (see §3.3.3). A cure for this problem is to give the instruction fetch stage a cache of its own, so that instructions can be loaded at the same time as data. Needless to say, the other cache is then called the **data cache.** In a cache hierarchy, the L2 cache may hold both data and instructions, in which case it is called a **unified cache.**

—

This brief discussion of memory caches by no means exhausts the topic. One problem, however, deserves special mention because it is so ubiquitous. The **cache-coherency problem** is that of insuring that the cache and main memory agree. A write through policy solves the problem nicely. With a write back policy, however, the cache block and corresponding memory block are different until the cache block is written back. This would make no difference if only the CPU were accessing memory. We shall see, however, that peripheral devices can read and write memory independently of the CPU, and care must be taken that nothing is read until the cache block has been written back. Interrupts and traps can also cause cache coherency problems.

4.3. Arrays and cache

Synopsis: Algorithms manipulating data in arrays are often regular in their memory references and can therefore be rearranged to perform well with cache systems. A standard technique is to arrange an algorithm so that it references memory with unit stride. Unfortunately, for the C family of languages unit stride means traversing rows, while for the Fortran family it means traversing columns. Thus we must distinguish between row-oriented and column-oriented algorithms.

Matrix-vector multiplication has a row-oriented algorithm based on AXPY and a column-oriented algorithm based on the dot product. Matrix-matrix multiplication can be reduced to matrix-vector multiplication. Although the resulting algorithms have good spatial locality of reference, they do not have good temporal locality. This problem can be alleviated by a technique called blocking — but only up to a point.

The full optimization of matrix operations is machine dependent and often requires a descent into assembly language code. The Basic Linear Algebra Subprograms (BLAS) are an interface (in Fortran or C) for performing operations from linear algebra. There are tailor-made, optimized versions of the BLAS for a number of machines. Thus programs that use the BLAS can run efficiently on any machine that has a library of optimized BLAS. The BLAS have three levels: vector operations, vector-matrix operations, and matrix-matrix operations, and subsets of them are specified for dense, banded, and sparse matrices. In cases where there are no optimal BLAS, the ATLAS system can generate them automatically.

—

```
    a     a + 40   a + 80   a + 120
 A(1,1)  A(1,2)   A(1,3)    A(1,4)

  a + 8   a + 48   a + 88   a + 128
 A(2,1)  A(2,2)   A(2,3)    A(2,4)

 a + 16   a + 56   a + 96   a + 136
 A(3,1)  A(3,2)   A(3,3)    A(3,4)

 a + 24   a + 64  a + 104   a + 144
 A(4,1)  A(4,2)   A(4,3)    A(4,4)

 a + 32   a + 72  a + 112   a + 152
 A(5,1)  A(5,2)   A(5,3)    A(5,4)
```

Figure 4.1: Column-major storage of an array.

Many of the algorithms in scientific computing are irregular in their memory references and hence difficult to fine tune for optimal use of cache. But algorithms involving data in arrays form an exception. These algorithms are often quite regular, and by rearranging the order of computation one can improve their cache performance. In this subsection we will use matrix-vector and matrix-matrix multiplication to illustrate some of the tricks of the trade.

4.3.1. Row and column orientation of algorithms

One technique for insuring good cache utilization is to write algorithms that access the elements of an array with unit stride. But here we come up against an unfortunate inconsistency between the major programming languages. C and C++ store two dimensional arrays in row-major order, as illustrated in Figure 3.5. The Fortran 77 and Fortran 95, on the other hand, store them in **column-major order** as in Figure 4.1. It is assumed that `A` is an array of double words beginning at address `a`. Thus to achieve unit strides an algorithm in the `C` family must proceed along the rows of its arrays, while an algorithm in the Fortran family must proceed along the columns. We call the former a **row-oriented algorithm** and the latter a **column-oriented algorithm**

To illustrate the two types of algorithms we consider the problem of computing

$$y = Ax, \tag{4.3}$$

where A is a matrix of order n and x and y are n-vectors. We will begin with the column-oriented algorithm.

Partition `A` in (4.3) by columns to get

$$y = (a_1\ a_2\ \cdots\ a_n)\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = x_1a_1 + x_2a_2 + \cdots + a_nx_n.$$

This leads to the following algorithm

```
1. y = 0;
2. for j = 1:n
3.     y = y + x(j)*A(:,j);
4. end
```
(4.4)

If we expand this into scalar form, we get

```
for i=1:n
  y(i) = 0;
end
for j=1:n
   for i=1:n
     y(i) = y(i) + x(j)*A(i,j);
   end
end
```

From this we see that the algorithm traverses `A` column by column. Incidentally, it is worth noting that line 3 in (4.4) is an example our old friend AXPY.

To derive the row-oriented algorithm, we partition A by rows to get

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} a_1^{\mathrm{T}} \\ a_2^{\mathrm{T}} \\ \vdots \\ a_n^{\mathrm{T}} \end{pmatrix} x = \begin{pmatrix} a_1^{\mathrm{T}}x \\ a_2^{\mathrm{T}}x \\ \vdots \\ a_n^{\mathrm{T}}x \end{pmatrix}$$

Thus we can write the following code.

```
for i=1:n
   y(i) = A(i,:)'*x;
end
```

In scalar form we have

```
for i=1:n
  y(i) = 0;
  for j=1:n
     y(i) = A(i,j)*x(j);
   end
end
```

From which it is seen that this algorithm traverses `A` by rows. Just as our column-oriented code was based on an AXPY, our row oriented code is based on a dot product.

We next consider the problem of matrix-matrix multiplication — specifically the computation of

$$C = AB,$$

where A, B, and C are of order n. We will treat only the column-oriented algorithm, leaving the row-oriented algorithm for the exercises.

The algorithm is easily derived. Partition C and B by columns to get

$$(c_1\ c_2\ \cdots\ c_n) = A(b_1\ b_2\ \cdots\ b_n) = (Ab_1\ Ab_2\ \cdots\ Ab_n).$$

Thus we can apply the column-oriented AXPY algorithm (4.4) to compute the columns $c_k = Ab_k$ $(k = 1, 2, \ldots, n)$. Here is the algorithm.

```
for k=1:n
  C(:,k) = 0;
  for j=1:n
     for i=1:n
        C(i,k) = C(i,k) + A(i,j)*B(j,k);
     end
  end
end
```

(4.5)

4.3.2. Spatial and temporal locality: Blocking

Algorithm (4.5) exhibits good locality of reference in the sense that consecutive references to memory are near each other in the address space. Such locality is called **spatial locality** or **locality in space.** This is in contrast to **temporal locality** or **locality in time,** where once a datum is in memory it is reused — preferably to the point where it is not needed again. Our algorithm exhibits poor locality in time, since it makes n passes over the matrix A as it compute Ab_i $(i = 1, \ldots, n)$. If A does not fit in cache, there is a real danger that each pass over A will generate a flurry of cache misses.

```
 1.  for k=1:m:n
 2.     kk = k+m-a;
 3.     C(:,k:kk) = 0;
 4.     for j=1:m:n
 5.        jj = j+m-1;
 6.        for i=1:m:n
 7.           ii = i+m-1;
 8.           C(i:ii,k:kk) = C(i:ii,k:kk)}
                                 + A(i:ii,j:jj)*B(j:jj,k:kk);
 9.        end
10.     end
11.  end
```

Figure 4.2: Blocked matrix multiplication

We can attempt to improve temporal equality by a technique known as **blocking.** Choose an integer m (which for ease of exposition we will assume divides n) and partition the product $C = AB$ in the form

$$\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1p} \\ C_{21} & C_{22} & \cdots & C_{2p} \\ \vdots & \vdots & & \vdots \\ C_{p1} & C_{p2} & \cdots & C_{pp} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & & \vdots \\ B_{p1} & B_{p2} & \cdots & B_{pp} \end{pmatrix},$$

where the blocks are all $m \times m$ and hence $p = n/m$. We now apply our column-oriented algorithm to this blocked version. The result is shown in Figure 4.2. If we assume m is such that the blocks in line 8 can be contained in cache, then it does not matter how we form the matrix product `A(i:ii,j:jj)*B(j:jj,k:kk)`.

This blocked algorithm reduces the number of passes over A to $p = n/m$. In other words, each time a block of A is brought into memory it is used to compute m columns of C, thus increasing temporal locality. This suggests that we should take m as large as possible. But the requirement that the blocks be in cache restricts the size of m. Typically, as the blocking parameter m increases, starting from 1, we will first see an improvement in performance followed by a deterioration. Given the complexities of the cache, especially when there are two or more levels, the optimal block size must be determined empirically.

4.3.3. The BLAS

This section and its predecessor make it abundantly clear that optimizing code is not a trivial undertaking. Performance depends on the execution pipeline, the scheduling of instructions, the properties of the cache memory, and so on. There is no law that says that improving one aspect of the code will not cause a deterioration in some other aspect. For the most part, a combination of careful coding and an aggressive optimizing compiler will produce satisfactory performance. But for the matrix operations treated in the last subsection, the only way to get optimal performance is to write machine dependent code at the assembly language level.

Fortunately, the number of matrix operations is restricted, so that it is possible for a computer manufacturer to produce at reasonable cost a package of optimized programs to perform them. The **basic linear algebra subprograms (BLAS)** provide a standard interface for these programs. This means that a program using the BLAS can obtain optimal performance on any machine for which there is a optimal set of BLAS.

It would take us too far afield to treat the BLAS in any great detail, but a brief description their organization is in order. The standard covers the languages Fortran 77, Fortran 95, and C. Subsets of the BLAS are specified for dense matrices, banded matrices, and sparse matrices. The BLAS also provides for mixed- or extended-precision arithmetic.

The BLAS are divided into three levels.

- **The level 1 BLAS.** These BLAS perform vector operations. We have already met AXPY and DOT — two computational workhorses. In addition, the level 1 BLAS provide routines to copy and scale vectors and to compute their norms.

- **The level 2 BLAS.** These BLAS perform matrix-vector operations — most notably the calculation of matrix-vector products treated in the last section. The also provide routines for solving triangular system and adding rank-one updates.

- **The level 3 BLAS.** These BLAS perform matrix-matrix operations. The most important of these is matrix-matrix multiplication. Optimized BLAS at this level use every trick of the trade to enhance their performance.

—

If you cannot find vendor supported BLAS for a particular machine, there are two alternatives. First, there is an unoptimized Fortran 77 reference set available at netlib. Second, the **ATLAS** system (Automatically Tuned Linear Algebra Software) will produce nearly optimal BLAS for a system

4.4. Virtual memory

Synopsis: When the data, including instructions, required by a process becomes so voluminous that it overflows main memory, it must be moved to a secondary store — usually a disk. Virtual memory treats main memory as a cache for the secondary store. The address space of the process — now called the virtual address space — is divided into blocks of equal size called pages and main memory is divided into page frames of the same size. When the CPU references a virtual address, a memory management unit looks for a page frame containing it. If it is found (a page hit) the reference is honored from main memory. Otherwise (a page fault) the page containing the reference is swapped in from disk.

Disks are very slow, and this slowness allows the CPU time to do things that could not be done in the cache memory system. Virtual memory systems are fully associative, pages are large, and replacement strategies are elaborate. Like cache systems, virtual memory depends on locality of reference for its success, but on a much larger scale.

The central problem of virtual memory is to translate virtual addresses into physical addresses in main memory. This must be done faster than a cache reference. One approach is to maintain a table of all pages, from which the physical address can be computed. To avoid the time required to consult a page table, a small number of recently used page table entries are cached in a translation look-aside buffer on the CPU. With the large address spaces on current machines, the page table becomes too large to maintain. The solution is to maintain page entries only for pages that are actually being used in the virtual memory.

The pages of a process are maintain on disk in a swap area (a.k.a. swap space). Pages are associated with addresses of a process's address space by memory mapping. Typically, memory mapping is done statically at load time when the process is started. However, it can be done dynamically at run time to give the process more memory. Shared memory can be created by mapping pages on disk of more than one process.

Virtual memory can be used to restrict access to parts of virtual memory. Typical permissions are privileged, read-write, write-only, and copy-on-write. Virtual memory can also be used to aid in loading a program for execution and in implementing shared libraries.

In scientific computing virtual memory will allow a program whose data exceeds the capacity of main memory to run — but not necessarily efficiently. With important tasks it may pay to code an out-of-core version that sidesteps the virtual memory systems and perform explicit data transfers between disk and main memory.

—

4.4.1. Basic concepts

The address space of a typical machine is far too large to be contained in main memory. On the other hand, secondary storage devices can hold very large amounts of data. This suggests that we map the machine's address space to secondary store and use main memory as if it were a cache.

To see how this is done, let us forget about main memory for the moment and consider processes and secondary stores. In a virtual memory system, each process has its own **virtual address space.** In principle, it could be identical with the address space of the machine, but it practice it is restricted to a subset. The data corresponding to this virtual address space is contained on a secondary store, which is usually a disk. The part of the disk corresponding to the virtual address space of a process is called its **swap space,** for reasons that will become clear later. Since there is a one-one correspondence between addresses in the process's virtual memory and items in the swap space, we say that the swap space has been **mapped** to the virtual address space of the process.

Now if there were no main memory, each memory reference by a process would have to be honored from the disk. But disk transactions are glacially slow compared to the speed of the CPU, and no program could expect to execute in a reasonable time under this regime. The cure is to use main memory as a cache for the data on disk.

Specifically, the virtual address space of the process, is divided up into **pages** of some fixed length (typically 8 KB to 4 MB). Main memory is divided into **pages frames** of the same length. In the parlance of caches, page frames correspond cache blocks and pages to blocks in main memory. All instructions that reference memory use virtual addresses. When a process makes a memory reference, the machine looks for a page frame containing the reference in main memory. This process known as **address translation** because, if it succeeds, it translates the virtual address into an address in main memory. If such a frame exists — an event called a **page hit** — the reference is honored from main memory. If not — a **page fault** — the page is located on the secondary storage device and swapped into a page frame in main memory, where it is used to honor the reference. The relations between the virtual address space, main memory, and disk are summarized in Figure 4.3.

All this is very cache-like. But there are differences. In the first place, the terminology is different — blocks and misses for cache become pages and faults for virtual memory. Second, each process has its own virtual address space and its own swap space on disk. Since the virtual address spaces of processes can overlap — and usually do — the target of an address in virtual memory depends on which process uses it. On the other hand, in the cache system, the target of an address is a fixed location in main memory, whatever its source. But perhaps the most substantial differences are due to the slowness of the secondary storage, to which we now turn.

4.4.2. Consequences of disk backup

The secondary storage device containing the virtual memory is invariably a disk. Now disks, like pipelines, have a latency — the time it takes to set up a read or write. Disk latency varies, but 10 milliseconds is not untypical. After the latency period, the disk can transfer informations at a rate of, say, 10 MB/sec. To get a feel for what these

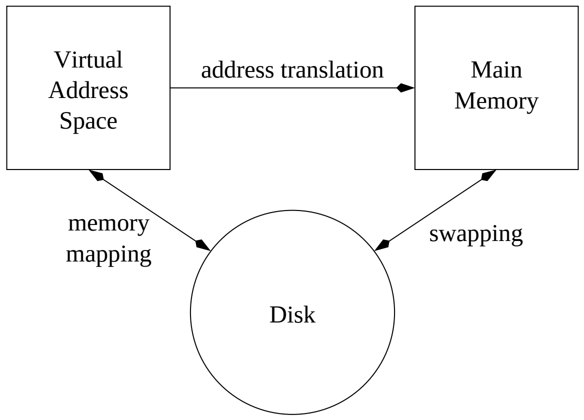

Figure 4.3: Relations between the virtual address space, main memory, and disk

numbers mean, suppose that a CPU has a cycle of 1 ns and call it a CPU second. Then a latency of 10 ms represents 10^7 CPU seconds or almost four CPU months!

This disparity between the CPU cycle and the latency of the disk has important consequences for any virtual memory system. The first is that the operating system can spare a CPU day or so to service a page fault in software. This is in contrast to a cache miss, which must be serviced quickly in hardware. A second consequence, is that the operating system may suspend the process that generated the page fault while the fault is being serviced. All this gives the operating system scope to do things that are impossible with cache memory. For example, virtual memory systems are fully associative — a page of virtually memory can, in principle, occupy any page frame in main memory. Again, there is time to implement elaborate replacement strategies.

Another consequence is that pages can be large. On the disk described above, a 100 KB page can be transferred in a period of time equal to the latency of the disk. In fact, one might say that pages *must* be large, so that the long disk latency can be played off against transmission time

Given the large penalty for a page fault, it is surprising that virtual memory systems work at all. To keep a cached system in balance, the miss rate must be inversely proportional to the miss penalty. Thus a virtual memory system must have a page-fault rate that is far smaller than cache miss rate. Of course, the page-fault rate depends on the pattern of memory references of the process in question. But it turns out that most processes spend most of their time referencing a **working set** of pages of virtual memory that can be contained in main memory. The working set may change over time, but typically the the change is slow enough so that page swaps do not overwhelm the computations. The large page size helps here. When a page fault occurs, the large new page gives locality of reference a chance too exercise its beneficial effects.

Latency is not the only limitation of disk storage. Although present-day disks have large capacities, they are not unlimited. In particular, a 64 bit address space — increasingly common in today's computers — amounts to $2^{64} \cong 1.8{\cdot}10^{19}$ bytes or 2 billion GBb, which is far beyond capacity of any currently conceivable disk. This means that only a small fraction of the virtual address space is available to any process. Paging can provide memory in excess of the size of main memory, but only to a limited degree.

4.4.3. Address translation

The central problem in designing a virtual memory system is **address translation.** Each memory reference by a process has a virtual address. This address must be converted to the physical address corresponding to the page frame containing the data (provided, of course, there is no page fault). Address translation has to be very fast, since the memory transaction must take place in a CPU cycle or two. Thus address translation is the most critical part of the virtual memory system.

The first problem is to decide at what point in the memory transaction the address translation takes place. One possibility, is to use virtual addresses in both the CPU and cache, and to translate only when the caching system must reference main memory. This is illustrated in the following diagram.

$$\boxed{\text{CPU}} \Longleftrightarrow \boxed{\text{Cache}} \Longleftrightarrow \boxed{\text{Address translation}} \Longleftrightarrow \boxed{\text{Main memory}}$$

This arrangement (called a **virtually addressed cache**) has the seeming advantage of avoiding address translation for most memory transactions, namely, those that are honored in cache. But it has difficulties. For example, when two processes share a block of memory, they may refer to the block by different virtual addresses — a problem called **aliasing.** Special care must be taken to see that aliased virtual addresses end up at the same location in the virtual cache. Problems like this are not insurmountable, but most machines use **physically addressed caches,** for which address translation takes place between the CPU and cache, as shown below.

$$\boxed{\text{CPU}} \Longleftrightarrow \boxed{\text{Address translation}} \Longleftrightarrow \boxed{\text{Cache}} \Longleftrightarrow \boxed{\text{Main memory}}$$

The next problem is how address translation is performed. The traditional way is based on **page tables.** To show how a page table works, suppose that we have a 32 bit virtual address space and pages of length 2^{18} bytes (256 KB). Thus there are $2^{32}/2^{14} = 2^{14}$ (16,384) pages. If, as in cache memory, we divide virtual address

$$\underbrace{\texttt{tttttttttttttt}}_{14}\underbrace{\texttt{oooooooooooooooooo}}_{18}$$

into a 14 bit tag field and an 18 bit offset, then the tag field can serve as a page number that indexes an entry into the page table. A typical page table entry will have the form

frame address	flags

where the frame address points to the beginning of the page frame in physical memory. Thus the physical address corresponding to the virtual address is

$$\text{frame address} + \text{offset}.$$

The flag field contains information about the page. It invariably contains a **valid bit** that says whether the page in question is in main memory. It might also contain a permission field that limits access access to the page (see §4.4.6).

A memory transaction is managed by a hardware component of the CPU called the **memory management unit (MMU)**. The MMU extracts the tag from the virtual address and examines the page table entry. If the page is valid, it adds the offset to the page address to get the physical address and goes on to complete the memory transaction. If the page is not valid, the MMU traps to the page-fault handler.

The trouble with page tables is that they are slow. They require one memory reference to get the physical address and another to complete the memory transaction. Moreover, references to the page table compete with regular memory references for space in cache memory. The cure is to maintain a small **translation look-aside buffer (TLB)** that contains page table entries for recently used pages. When a memory reference occurs, the MMU first looks in the TLB for the page table entry. If it is there, the MMU completes the memory transaction forthwith. Otherwise, the MMU gets the entry from the page table. In either case, if the page is invalid, the MMU traps to the page-fault handler. Whenever the MMU is forced to get a page table entry from the page table, that entry is moved to the TLB, overwriting one of the current TLB entries.

The combination of TLB and page table works well enough when the virtual address space is sufficiently small. But in our example, if we up the address space from 32 to 64 bits and keep our 256 KB pages, the number of pages becomes is on the order of 7 trillion. A page table of this size is impossible to maintain. The cure is to store page entries only for pages that are actually in use — a set that is far smaller that the set of all pages. Note that 'page table entry' has become 'page entry' since we have abandoned page tables. The following is a brief description of the page-entry system on the SUN ULTRA.[6]

The translation process has three levels. The first is a TLB. If the TLB fails, the MMU traps to the operating system, which attempts to find the page entry in a translation storage buffer (TSB). This is a direct mapped cache in main memory

[6]Adapted from Tannenbaum.

containing more page entries. If the page is not found there, the operating system looks in a third level table for the page entry. This table is entirely the responsibility of the operating system, and it can take any form. Any time a page entry is found in a lower level, it is promoted to the higher levels.

Note that the failure to find a page entry corresponding to a virtual address is not a page fault. Rather it is a fatal error, since all existing pages are supposed to have entries at some level in the hierarchy of buffers.

4.4.4. Replacement strategies

When a page fault occurs, the page must be brought from disk into a page frame, and it replaces of the page currently in the frame. Of course, if the old page has been written — generally indicated by a dirty bit in the flag field of page entry — it must written back to disk before the new page can be swapped in. The customary rule for deciding which page frame to choose is LRU (least recently used). A strict LRU strategy is too expensive to implement, but there are satisfactory approximations.

♠ Exercises on LRU.

4.4.5. Virtual memory and shared memory

Virtual memory provides a way for processes to communicate with one another by means of shared blocks of memory. The basic idea is to map pages on the disk to the address space of several processes. This technique is illustrated in Figure 4.4, which shows the address spaces of two processes, A and B, and of main memory. The subdivisions of these spaces represent pages. The hatched area is the shared memory block, and the arrows indicate the mapping of the pages onto main memory. Main memory is connected to the disk to remind us that this is a paging environment and that pages of shared memory also move in and out of main memory.

When A makes a reference to a page in the shared memory the address translation procedure follows the corresponding arrow to main memory, where the reference is honored. The same is true of B. Thus A and B can communicate with one another by writing and reading their common shared memory.

There are three comments to make about shared memory.

- The implementation of shared memory is in principle simple. If the shared memory is new, the operating system must find swap space on the disk for it. New or old, the operating system then maps the pages on disk to the address space of of A and B (and C and D, and so on, if there are more than two processes sharing the block of memory). After the pages have been mapped, the virtual memory system does all the work. In practice the user must call system routines to create and access shared memory. The details vary from system to system. In Unix systems there are two different flavors of

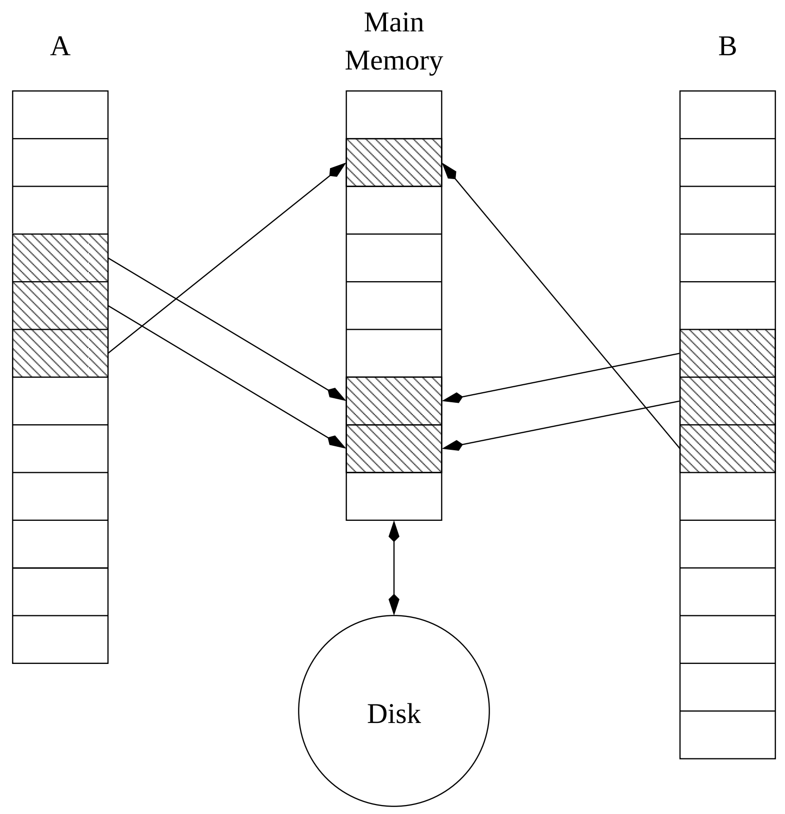

Figure 4.4: Shared memory: Address spaces and mappings

shared memory, System V IPC and BSD `mmap`, both of which are C based. Fortran facilities for shared memory are at best primitive.

• The shared memory can lie in different parts of the address space of A and B, as is indicated in Figure 4.4. This feature is important, since A and B may be two otherwise unrelated processes that cannot communicate to agree on a common address for the block.

• When several processes are reading and writing a shared block of memory, care must be taken that these operations occur in the intended order. This is called **synchronization,** and we will treat it in §????.

4.4.6. Control of memory access

A perennial problem in managing processes is the control their access to memory. For example, it would not do for a process to access memory reserved for the operating system. Virtual memory provides an elegant solution for this problem. Permission bits can be associated with each page. When a page is referenced, the MMU can check to see if the process in question is permitted to perform the requested operation on the page. If so, the process proceeds as usual. If not, the MMU traps to the operation system, which terminates the process with an error message. On Unix systems this is the notorious "segmentation error."

The following are among the more common permissions.

- **Privileged.** Only the operating system may access this page.

- **Read-write.** The process may read and write the page in question.

- **Read-only.** The process may only read the page. Any attempt to write it will result in an error message. This is the status of many pages in shared libraries (see §9.2).

- **Copy-on-write.** This option is useful in shared memory applications. The page in question is treated as read-only. But when a process tries to write, instead of trapping the operating system creates a copy of the page and maps it to the corresponding address in the address space of the process. Thereafter, all reads and writes by that process are directed to the new page, while the original remains unchanged. This permission can be useful in managing shared libraries.

4.4.7. Some afterthoughts on virtual memory

Although we introduced virtual memory as a foil against the limited capacity of main memory, it actually solves some other problems in process management. We have already seen how it can be used to support shared memory and protect memory from unauthorized access. We will see later that it can also be used to load processes into memory and set them running (§9.1.1). Virtual memory is also used to support shared libraries (§9.2).

Finally, virtual memory is not always the best solution to the problem of memory management. In scientific computing it is easy for the data processed by an algorithm to exceed the capacity of main memory. In such a case, virtual memory will insure that the algorithm will run to completion. But it does not insure that it will run efficiently. If the algorithm is a widely used one, it may pay to design an **out of core** version.[7] Such algorithms perform explicit transfers to and from disk and generally use different

[7]This phrase is a legacy of the time when random access memory was built of tiny ferrite doughnuts, which could be magnetized in two directions — one representing a zero and the other a one. The doughnuts were called **cores** and the memory **core memory** or simply core for short.

data structures than their in-core counterparts. In this way they are able to optimize the transfers of data and overlap them with numerical computations.

EXERCISES

1. A byte-addressable machine has 32 bit address space and a 16KB cache with a block size of 64B. Give the size of the tag, index, and offset fields for the following configurations.

   1. Direct mapped cache
   2. 4-way set-associative cache
   3. 2^k set-associative cache

2. This problem investigates the effect of block size on cache performance. We suppose that

$$\mathrm{T_a} = \mathrm{T_h} + \mathrm{P_m T_m},$$

Where $\mathrm{T_a}$ is the average access time, $\mathrm{T_h}$ is the access time for a hit, $\mathrm{P_m}$ is the probability of a miss and $\mathrm{T_m}$ is the miss penalty. Let n be the block size. Assume that

$$\mathrm{P_m} = \frac{\mathrm{c}}{\sqrt{\mathrm{n}}};$$

i.e., the probability of a miss decreases slowly as the block size increases. Assume further that

$$\mathrm{T_m} = \mathrm{a} + \mathrm{nb}.$$

Here a is the latency of the memory and b is the transfer rate once the memory has been engaged. Assume, as is generally true, that $\mathrm{a} > \mathrm{b}$. Show that there is a unique positive value of n that minimizes $\mathrm{T_a}$ and determine its value in terms of a, b, and c. Explain in plain words what is going on. What happens when we model $\mathrm{P_m}$ by

$$\mathrm{P_m} = \frac{\mathrm{c}}{\mathrm{n}}?$$

For purposes of this exercise you may assume that n is a continuous variable, so that you can use elementary calculus to find minima.

3. The following mathematical exercise relates to the repetition of references with constant stride in banked memorys. Let m and a be positive integers. Then

$$n \bmod m$$

is the remainder after the division of n by m.

   1. Let s be a positive integer and consider the sequence

$$n_i = is \bmod m, \qquad i = 0, 1, 2, \ldots.$$

   Since $0 \le n_i < m$, there must be a smallest positive k such that $n_k = n_j$ for some $j < k$. Show that $n_k = 0$.

2. Two positive integers are said to be relatively prime if they have no common factors. Let k be the smallest positive integer such that $n_k = 0$. Show that $k = m$ if and only if m and k are relatively prime. [Hint: Use the fact that any positive integer greater than one can be uniquely factored into a produce of primes.]

4. For the purposes of this exercise, assume a CPU with a single, direct-mapped cache. Let `A` be an $n \times n$ byte array stored columnwise and suppose that n is greater than the number of blocks and the block size. Determine the behavior of the cache when the entire array is accessed columnwise. Do the same for rowwise access. Assume you are starting with an uninitialized cache — a.k.a. a cold cache. [Hint: See Exercise 4.3.]

5. Let L be an upper triangular matrix of order n. Give row and column oriented Matlab functions

```
x = rowltsolve(L, b)
x = colltsolve(L, b)
```

for solving the system $Lx = b$. Wherever possible, prefer vector operations to scalar operations (this is equivalent to using level one BLAS in Fortran). Use the following script to make sure your functions work.

```
n = 4;
L = tril(randn(n));
b = L*ones(n,1);
rowltsolve(L, b)
colltsolve(L, b)
```