21

The Memory Hierarchy

mem-o-ry: a device in which information can be inserted and stored and from which it may be extracted when wanted
hi-er-ar-chy: a graded or ranked series

Webster's Dictionary

An idealized *random access memory* (RAM) has N words indexed by integers such that any word can be fetched or stored – using its integer address – equally quickly. Hardware designers can make a big slow memory, or a small fast memory, but a big fast memory is prohibitively expensive. Also, one thing that speeds up access to memory is its nearness to the processor, and a big memory must have some parts far from the processor no matter how much money might be thrown at the problem.

Almost as good as a big fast memory is the combination of a small fast *cache memory* and a big slow *main memory*; the program keeps its frequently used data in cache and the rarely used data in main memory, and when it enters a phase in which datum x will be frequently used it may move x from the slow memory to the fast memory.

It's inconvenient for the programmer to manage multiple memories, so the hardware does it automatically. Whenever the processor wants the datum at address x, it looks first in the cache, and – we hope – usually finds it there. If there is a *cache miss* – x is not in the cache – then the processor fetches x from main memory and places a copy of x in the cache so that the next reference to x will be a *cache hit*. Placing x in the cache may mean removing some other datum y from the cache to make room for it, so that some future access to y will be a cache miss.

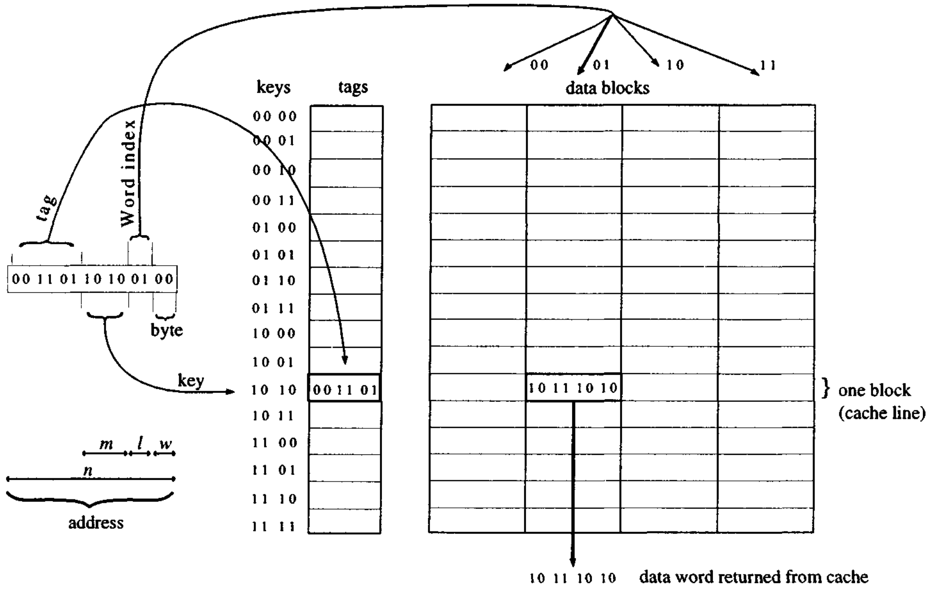

FIGURE 21.1. Organization of a direct-mapped cache. *Key* field of the address is used to index the *tags* array and the *data blocks*; if *tags*[*key*] matches the *tag* field of the address then the data is valid (cache hit). *Word index* is used to select a word from the cache block.

21.1 CACHE ORGANIZATION

A *direct-mapped* cache is organized in the following way to do this quickly. There are 2^m blocks, each holding 2^l words of 2^w bytes; thus, the cache holds 2^{w+l+m} bytes in all, arranged in an array *Data*[*block*][*word*][*byte*]. Each block is a copy of some main-memory data, and there is a *tag* array indicating where in memory the current contents come from. Typically, the word size 2^w might be 4 bytes, the block size 2^{w+l} might be 32 bytes, and the cache size 2^{w+l+m} might be as small as 8 kilobytes or as large as 2 megabytes.

tag	*key*	*word*	*byte*
$(n-(m+l+w))$ bits	m bits	l	w

Given an address x, the cache unit must be able to find whether x is in the cache. The address x is composed of n bits, $x_{n-1}x_{n-2}\ldots x_2x_1x_0$ (see Figure 21.1). In a *direct-mapped* cache organization, we take the middle bits as the *key* $= x_{w+l+m-1}x_{w+l+m-2}\ldots x_{w+l}$, and hold the data for x in *Data*[*key*].

The high bits $x_{n-1}x_{n-2}\ldots x_{w+l+m}$ form the *tag*, and if *Tags*[*key*] $\neq$ *tag* then there is a *cache miss* – the word we require is not in cache. In this case contents of *data*[*key*] are sent back to main memory, and the contents of memory at address $x_{n-1}\ldots x_{w+l}$, are fetched into the kth cache block (and also sent to the CPU). Access time for main memory is much longer than the cache access time, so frequent misses are undesirable.

The next time address x is fetched, if no intervening instruction has fetched another address with the same key but different tag, there will be a *cache hit*: *Tags*[*key*] $=$ *tag*, and bits $x_{w+l-1}\ldots x_w$ will address a word within the keyth block: the contents of *data*[*key*][$x_{w+l-1}\ldots x_w$] are transferred to the processor. This is much faster than going all the way to main memory for the data. If the fetching instruction is a byte-fetch (instead of a word-fetch), then (typically) the processor takes care of selecting the byte $x_{l-1}\ldots x_0$ from the word.

Another common organization is the *set-associative* cache, which is quite similar but can hold more than one block with the same *key* value. The compiler optimization strategies presented in this chapter are valid for both direct-mapped caches and set-associative caches, but they are a bit more straightforward to analyze for direct-mapped caches.

Write-hit policy. The paragraphs above explain what happens on a *read*, when the CPU asks for data at address x. But what happens when the CPU writes data at address x? If x is in the cache, this is a *write hit*, which is easy and efficient to process. On a write hit, main memory may be updated now (write-through), or only when the cache block is about to be flushed from the cache (write-back), but choice of write-hit policy does not much affect the compilation and optimization of sequential programs.

Write-miss policy. If the CPU writes data at an address not in the cache, this is a *write miss*. Different machines have different write-miss policies:

Fetch-on-write. Word x is written to the cache. But now the other data words in the same cache block belonged to some other address (that had the same *key* as x), so to make a valid cache block the other words are fetched from main memory. Meanwhile, the processor is stalled.

Write-validate. Word x is written to the cache. The other words in the same cache block are marked *invalid*; nothing is fetched from main memory, so the processor is not stalled.

Write-around. Word x is written directly to main memory, and not to the cache. The processor is not stalled, as no response is required from the memory sys-

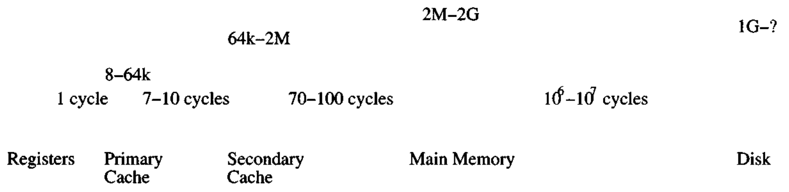

FIGURE 21.2. The memory hierarchy.

tem. Unfortunately, the next time *x* is fetched there will be a read miss, which will delay the processor.

The write-miss policy can affect how programs should be optimized (see pages 503 and 508).

Several layers of cache. A modern machine has a *memory hierarchy* of several layers, as shown in Figure 21.2: inside the processor are *registers*, which can typically hold about 200 bytes in all and can be accessed in 1 processor cycle; a bit farther away is the *primary cache*, which can typically hold 8–64 kilobytes and be accessed in about 2–3 cycles; then the *secondary cache* can hold about a megabyte and be accessed in 7–10 cycles; *main memory* can hold 100 megabytes and be accessed in 100 cycles. The primary cache is usually split into an *instruction cache* – from which the processor fetches instructions to execute, and a *data cache*, from which the processor fetches and stores operands of instructions. The secondary cache usually holds both instructions and data.

Many processors can issue several instructions per cycle; the number of *useful* instructions in a cycle varies, depending on data-dependence and resource constraints (see page 469), but let us suppose that two useful instructions can be completed in each cycle, on the average. Then a primary-cache miss is a 15-instruction delay (7–10 cycles, times 2), and a secondary-cache miss is a 200-instruction delay.

This cache organization has several consequences of interest to the programmer (and often to the compiler):

Byte fetch: Fetching a single byte is often more expensive than fetching a whole word, because the memory interface delivers a whole word at a time, so the processor must do extra shifting.

Byte store: Storing a single byte is *usually* more expensive than storing a whole word, because the other bytes of that word must be fetched from the cache and stored back into it.

Temporal locality: Accessing (fetching or storing) a word that has been recently accessed will usually be a cache hit.

Spatial locality: Accessing a word in the same cache-block as one that has been accessed recently will usually be a cache hit.

Cache conflict: If address a and address $a + i \cdot 2^{w+b+m}$ are both frequently accessed, there will be many cache misses because accessing one will throw the other out of the cache.

The compiler can do optimizing transformations that do not decrease the number of instructions executed, but that decrease the number of cache misses (or other memory stalls) that the program encounters.

21.2 CACHE-BLOCK ALIGNMENT

The typical cache-block size (B = about 8 words, more or less) is similar to the typical data-object size. We may expect that an algorithm that fetches one field of an object will probably fetch other fields as well.

If x straddles a multiple-of-B boundary, then it occupies portions of two different cache blocks, both of which are likely to be active at the same time. On the other hand, if x does not cross a multiple-of-B boundary, then accessing all the fields of x uses up only one cache block.

To improve performance by using the cache effectively, the compiler should arrange that data objects are not unnecessarily split across blocks.

There are simple ways to accomplish this:

1. Allocate objects sequentially; if the next object does not fit in the remaining portion of the current block, skip to the beginning of the next block.
2. Allocate size-2 objects in one area of memory, all aligned on multiple-of-2 boundaries; size-4 objects in another area, aligned on multiple-of-4 boundaries, and so on. This eliminates block-crossing for many common-sized objects, without wasted space between the objects.

Block alignment can waste some space, leaving unused words at the end of some blocks, as shown in Figure 21.3. However, the execution speed may improve; for a given phase of the program, there is a set S of frequently accessed objects, and alignment may reduce the number of cache blocks occupied by S from a number greater than the cache size to a number that fits in the cache.

Alignment can be applied both to global, static data and to heap-allocated data. For global data, the compiler can use assembly-language alignment directives to instruct the linker. For heap-allocated records and objects, it is not

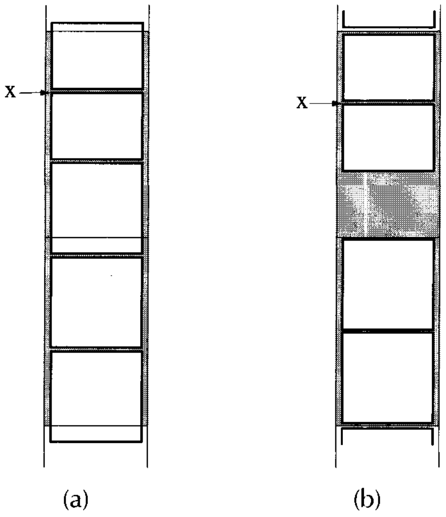

FIGURE 21.3. Alignment of data objects (or basic blocks) to avoid crossing cache-block boundaries is often worthwhile, even at the cost of empty space between objects.

the compiler but the memory allocator within the runtime system that must place objects on cache-block boundaries, or otherwise minimize the number of cache-block crossings.

ALIGNMENT IN THE INSTRUCTION CACHE

Instruction "objects" (basic blocks) occupy cache blocks just as do data records, and the same considerations of block-crossing and alignment apply to instructions. Aligning the beginning of frequently executed basic blocks on multiple-of-B boundaries increases the number of basic blocks that fit simultaneously in the instruction cache.

Infrequently executed instructions should not be placed on the same cache blocks as frequently executed instructions. Consider the program

```
P;
if x then Q;
R;
```

where x is rarely true. We could generate code for it in either of the ways shown in Figure 21.4; but placing Q out-of-line means that this series of statements (usually) occupies two cache blocks, but placing Q straddling cache blocks between P and R will mean that even in the common case, where Q

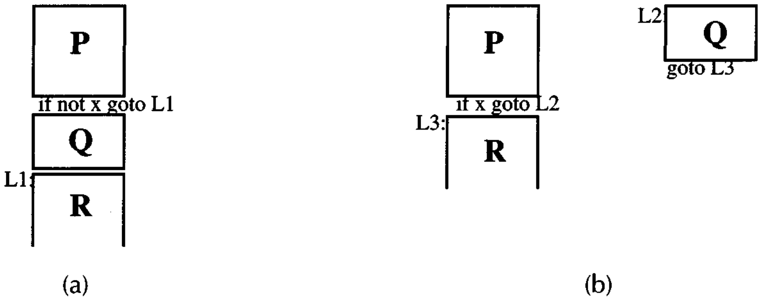

FIGURE 21.4. If x is rarely true, basic-block placement (a) will occupy three in-cache blocks, while (b) will usually occupy only two.

is not executed, this part of the program will occupy three blocks in the cache.

On some machines it is particularly important to align the target of a branch instruction on a power-of-2 boundary. A modern processor fetches an aligned block of k (2 or 4 or more) words. If the program branches to some address that is not on a multiple-of-k boundary, then the instruction-fetch is not fetching k useful instructions.

An optimizing compiler should have a *basic-block-ordering* phase, after instruction selection and register allocation. *Trace scheduling* (as described in Section 8.2) can then be used to order a frequently executed path through a contiguous set of cache blocks; in constructing a trace through a conditional branch, it is important to follow the most-likely-taken out-edge, as determined by *branch prediction* (as described in Section 20.3).

21.3 PREFETCHING

If a load instruction misses the primary (or secondary) cache, there will be a 7–10 cycle delay (or a 70–100 cycle delay, respectively) while the datum is fetched from the next level of the memory hierarchy. In some cases, the need for that datum is predictable many cycles earlier, and the compiler can insert *prefetch* instructions to start the fetching earlier.

A *prefetch* instruction is a hint to the hardware to start bringing data at address x from main memory into the cache. A prefetch never stalls the processor – but on the other hand, if the hardware finds that some exceptional condition (such as a page fault) would occur, the prefetch can be ignored. When *prefetch*(x) is successful, it means that the next load from x will hit the cache; an unsuccessful prefetch might cause the next load to miss the cache, but the program will still execute correctly. Many machines now have some form of prefetch instruction.

Of course, one reasonable alternative is – instead of starting the fetch earlier – to just delay the instruction that uses the result of the fetch until later, using the software-pipelining techniques described in Chapter 20. In fact, processors that dynamically reorder instructions (to account for operands not ready) achieve this effect without any special work by the compiler.

The problem with using software pipelining or dynamic rescheduling to hide secondary-cache misses is that it increases the number of live temporaries. Consider the following dot-product loop as an example:

```
$L_1 : x \leftarrow M[i]$
       $y \leftarrow M[j]$
       $z \leftarrow x \times y$
       $s \leftarrow s + z$
       $i \leftarrow i + 4$
       $j \leftarrow j + 4$
       if $i < N$ goto $L_1$
```

If the data for the i and j arrays are not in the primary cache, or if N is large (> 8 kilobytes or so) so that the arrays cannot possibly fit in the cache, then each time i or j crosses to a new multiple-of-B boundary (into a new cache block), there will be a cache miss. In effect, the miss rate will be exactly W/B, where W is the word size and B is the block size. Typical values for W/B are $\frac{1}{4}$ or $\frac{1}{8}$, and this is a rather high miss rate.

The penalty for a primary cache miss is perhaps 7 cycles, or (on a dual-instruction-issue-per-cycle machine) 14 instructions. This would stall the processor of an early-'90s machine for 14 instructions, but a good late-'90s machine with out-of-order execution will find some other instruction to execute that is not data-dependent on the load.

The effective order of execution, on a dynamic-instruction-reordering machine, is shown in Figure 21.5a. When $x_1 \leftarrow M[i_0]$ is fetched there is a cache miss, so instructions data-dependent on x_1 cannot be issued for 11 cycles. In the meantime, i_1 and j_1, and even i_2 and j_2 can be computed; and the fetch $x_2 \leftarrow M[i_1]$ can be issued.

As the number of uncompleted loop-iterations increases, the number of live or reserved registers increases proportionately. The cache-misses for x_2, x_3, x_4 are the *same* miss as for x_1 because they are all in the same cache

Cache Delay	Instruction issued	Live or reserved registers
	$x_1 \leftarrow M[i_0]$	$s_0 i_0 j_0 x_1$
│	$y_1 \leftarrow M[j_0]$	$s_0 i_0 j_0 x_1 y_1$
│ │	$i_1 \leftarrow i_0 + 4$	$s_0 i_1 j_0 x_1 y_1$
│ │	$j_1 \leftarrow j_0 + 4$	$s_0 i_1 j_1 x_1 y_1$
│ │	**if** $i_1 < N$ ...	$s_0 i_1 j_1 x_1 y_1$
│ │	$x_2 \leftarrow M[i_1]$	$s_0 i_1 j_1 x_1 y_1 x_2$
│ │	$y_2 \leftarrow M[j_1]$	$s_0 i_1 j_1 x_1 y_1 x_2 y_2$
│ │	$i_2 \leftarrow i_1 + 4$	$s_0 i_2 j_1 x_1 y_1 x_2 y_2$
│ │	$j_2 \leftarrow j_1 + 4$	$s_0 i_2 j_2 x_1 y_1 x_2 y_2$
│ │	**if** $i_2 < N$ ...	$s_0 i_2 j_2 x_1 y_1 x_2 y_2$
│ │	$x_3 \leftarrow M[i_2]$	$s_0 i_2 j_2 x_1 y_1 x_2 y_2 x_3$
│ │	$y_3 \leftarrow M[j_2]$	$s_0 i_2 j_2 x_1 y_1 x_2 y_2 x_3 y_3$
│	$i_3 \leftarrow i_2 + 4$	$s_0 i_3 j_2 x_1 y_1 x_2 y_2 x_3 y_3$
	$z_1 \leftarrow x_1 + y_1$	$s_0 i_3 j_2 z_1 x_2 y_2 x_3 y_3$
	$s_1 \leftarrow s_0 + z_1$	$s_1 i_3 j_2 x_2 y_2 x_3 y_3$
	$z_2 \leftarrow x_2 + y_2$	$s_1 i_3 j_2 z_2 x_3 y_3$
	$s_2 \leftarrow s_1 + z_2$	$s_2 i_3 j_2 x_3 y_3$
	$z_3 \leftarrow x_3 + y_3$	$s_2 i_3 j_2 z_3$
	$s_3 \leftarrow s_2 + z_3$	$s_3 i_3 j_2$
	$j_3 \leftarrow j_2 + 4$	$s_3 i_3 j_3$
	if $i_3 < N$ ...	$s_3 i_3 j_3$
	$x_4 \leftarrow M[i_3]$	$s_3 i_3 j_3 x_4$
	$y_4 \leftarrow M[j_3]$	$s_3 i_3 j_3 x_4 y_4$
	$z_4 \leftarrow x_4 + y_4$	$s_3 i_3 j_3 z_4$
	$s_4 \leftarrow s_2 + z_3$	$s_4 i_3 j_3$
	$i_4 \leftarrow i_3 + 4$	$s_4 i_4 j_3$
	$j_4 \leftarrow j_3 + 4$	$s_4 i_4 j_4$
	if $i_4 < N$ ...	$s_4 i_4 j_4$
	$x_5 \leftarrow M[i_4]$	$s_4 i_4 j_4 x_5$
│	$y_5 \leftarrow M[j_4]$	$s_4 i_4 j_4 x_5 y_5$
│ │	$i_4 \leftarrow i_3 + 4$	$s_4 i_5 j_4 x_5 y_5$

(a) Without prefetching

Cache Delay	Instruction issued	Live or reserved registers
	fetch $M[i_0 + 16]$	$s_0 i_0 j_0$
│	$x_1 \leftarrow M[i_0]$	$s_0 i_0 j_0 x_1$
│	$y_1 \leftarrow M[j_0]$	$s_0 i_0 j_0 x_1 y_1$
│	$z_1 \leftarrow x_1 + y_1$	$s_0 i_0 j_0 z_1$
│	$s_1 \leftarrow s_0 + z_1$	$s_1 i_0 j_0$
│	$i_1 \leftarrow i_0 + 4$	$s_1 i_1 j_0$
│	$j_1 \leftarrow j_0 + 4$	$s_1 i_1 j_1$
│	**if** $i_1 < N$ ...	$s_1 i_1 j_1$
│	$x_2 \leftarrow M[i_1]$	$s_1 i_1 j_1 x_2$
│	$y_2 \leftarrow M[j_1]$	$s_1 i_1 j_1 x_2 y_2$
│	$z_2 \leftarrow x_2 + y_2$	$s_1 i_1 j_1 z_2$
│	$s_2 \leftarrow s_1 + z_2$	$s_2 i_1 j_1$
	$i_2 \leftarrow i_1 + 4$	$s_2 i_2 j_1$
	$j_2 \leftarrow j_1 + 4$	$s_2 i_2 j_2$
	if $i_2 < N$ ...	$s_2 i_2 j_2$
	fetch $M[j_2 + 16]$	$s_2 i_2 j_2$
│	$x_3 \leftarrow M[i_1]$	$s_2 i_2 j_2 x_3$
│	$y_3 \leftarrow M[j_1]$	$s_2 i_2 j_2 x_3 y_3$
│	$z_3 \leftarrow x_2 + y_2$	$s_2 i_2 j_2 z_3$
│	$s_3 \leftarrow s_2 + z_2$	$s_3 i_2 j_2$
│	$i_3 \leftarrow i_2 + 4$	$s_3 i_3 j_2$
│	$j_3 \leftarrow j_2 + 4$	$s_3 i_3 j_3$
│	**if** $i_3 < N$ ...	$s_3 i_3 j_3$
│	$x_4 \leftarrow M[i_3]$	$s_3 i_3 j_3 x_4$
│	$y_4 \leftarrow M[j_3]$	$s_3 i_3 j_3 x_4 y_4$
│	$z_4 \leftarrow x_4 + y_4$	$s_3 i_3 j_3 z_4$
│	$s_4 \leftarrow s_3 + z_4$	$s_4 i_3 j_3$
	$i_4 \leftarrow i_3 + 4$	$s_4 i_4 j_3$
	$j_4 \leftarrow j_3 + 4$	$s_4 i_4 j_4$
	if $i_4 < N$ ...	$s_4 i_4 j_4$
│	**fetch** $M[i_4 + 16]$	$s_4 i_4 j_4$

(b) With prefetching

FIGURE 21.5. Execution of a dot-product loop, with 4-word cache blocks.

(a) Without prefetching, on a machine with dynamic instruction reordering, the number of outstanding instructions (reserved registers) grows proportionally to the cache-miss latency.

(b) With prefetching, the hardware reservation table never grows large. (Steady-state behavior is shown here, not the initial transient.)

block, so x_1, x_2, x_3, x_4 all become available at about the same time. Iterations 5–8 (which use the next cache block) would be dynamically scheduled like iterations 1–4, and so on.

The primary-cache latency, illustrated here, is usually small enough to handle without prefetching techniques. But with a secondary cache miss latency of 200 instructions (i.e. 29 loop iterations), there will be about 116 outstanding instructions (computations of x, y, z, s waiting for the cache miss), which may exceed the capacity of the machine's instruction-issue hardware.

Prefetch instructions. Suppose the compiler inserts a *prefetch* instruction for address a, in advance of the time a will be fetched. This is a hint to the computer that it should start transferring a from main memory into the cache. Then, when a is fetched a few cycles later by an ordinary load instruction, it will hit the cache and there will be no delay.

Many machines don't have a prefetch instruction as such, but many machines do have a non-blocking *load* instruction. That is, when $r_3 \leftarrow M[r_7]$ is performed, the processor does not stall even on a cache miss, *until r_3 is used as an operand of some other instruction.* If we want to prefetch address a, we can just do $r_t \leftarrow M[a]$, and then *never use the value of r_t*. This will start the load, bringing the value into cache if necessary, but not delay any other instruction. Later, when we fetch $M[a]$ again, it will hit the cache. Of course, if the computation was already *memory-bound* – fully utilizing the load/store unit while the arithmetic units are often idle – then prefetching using ordinary *load* instructions may not help.

If the computation accesses every word of an array sequentially, it uses several words from each cache block. Then we don't need to prefetch every word – just one word per cache block is enough. Assuming a 4-byte word and 16-byte cache block, the dot-product loop with prefetching looks something like this:

```
$L_1$ : if $i \bmod 16 = 0$ then prefetch $M[i + K]$
      if $j \bmod 16 = 0$ then prefetch $M[j + K]$
      $x \leftarrow M[i]$
      $y \leftarrow M[j]$
      $z \leftarrow x \times y$
      $s \leftarrow s + z$
      $i \leftarrow i + 4$
      $j \leftarrow j + 4$
      if $i < N$ goto $L_1$
```

```
$L_1$ : prefetch $M[i+K]$
      prefetch $M[j+K]$
      $x \leftarrow M[i]$
      $y \leftarrow M[j]$
      $z \leftarrow x \times y$
      $s \leftarrow s + z$
      $i \leftarrow i + 4$
      $j \leftarrow j + 4$
      **if** $i \geq N$ **goto** $L_2$
      $x \leftarrow M[i]$
      $y \leftarrow M[j]$
      $z \leftarrow x \times y$
      $s \leftarrow s + z$
      $i \leftarrow i + 4$
      $j \leftarrow j + 4$
      **if** $i \geq N$ **goto** $L_2$
      $x \leftarrow M[i]$
      $y \leftarrow M[j]$
      $z \leftarrow x \times y$
      $s \leftarrow s + z$
      $i \leftarrow i + 4$
      $j \leftarrow j + 4$
      **if** $i \geq N$ **goto** $L_2$
      $x \leftarrow M[i]$
      $y \leftarrow M[j]$
      $z \leftarrow x \times y$
      $s \leftarrow s + z$
      $i \leftarrow i + 4$
      $j \leftarrow j + 4$
      **if** $i < N$ **goto** $L_1$
$L_2$ :
```

```
$L_1$ : $n \leftarrow i + 16$
      **if** $n + K \geq N$ **goto** $L_3$
      prefetch $M[i+K]$
      prefetch $M[j+K]$
$L_2$ : $x \leftarrow M[i]$
      $y \leftarrow M[j]$
      $z \leftarrow x \times y$
      $s \leftarrow s + z$
      $i \leftarrow i + 4$
      $j \leftarrow j + 4$
      **if** $i < n$ **goto** $L_2$
      **goto** $L_1$
$L_3$ : $x \leftarrow M[i]$
      $y \leftarrow M[j]$
      $z \leftarrow x \times y$
      $s \leftarrow s + z$
      $i \leftarrow i + 4$
      $j \leftarrow j + 4$
      **if** $i < N$ **goto** $L_3$
```

PROGRAM 21.6. Inserting prefetches using loop unrolling or nested loops.

The value K is chosen to match the expected cache-miss latency. For a secondary-cache-miss latency of 200 instructions, when each loop iteration executes 7 instructions and advances i by 4, we would use $K = 200 \cdot 4/7$ rounded up to the nearest multiple of the block size; that is, about 128. Figure 21.5b uses prefetching to "hide" a cache latency of 11 instructions, so $K = 16$, the block size. An additional improvement that may be helpful on some machines, when K is small, is to avoid overlapping the prefetch latencies so the memory hardware needn't process two misses simultaneously.

In practice, we don't want to test $i \bmod 16 = 0$ in each iteration, so we unroll the loop, or nest a loop within a loop, as shown in Program 21.6. The loop-unrolled version on the left could be further improved – in ways unrelated to prefetching – by removing some of the intermediate **if** statements, as described in Section 18.5.

Prefetching for stores. Sometimes we can predict at compile time that a *store* instruction will miss the cache. Consider the following loop:

```
for $i \leftarrow 0$ to $N - 1$
    $A[i] \leftarrow i$
```

If the array A is larger than the cache, or if A has not recently been accessed, then each time i crosses into a new cache block there will be a write miss. If the write-miss policy is *write-validate*, then this is no problem, as the processor will not be stalled and all the marked-invalid words will be quickly overwritten with valid data. If the policy is *fetch-on-write*, then the stalls at each new cache block will significantly slow down the program. But prefetching can be used here:

```
for $i \leftarrow 0$ to $N - 1$
    if $i$ mod $blocksize = 0$ then prefetch $A[i + K]$
    $A[i] \leftarrow i$
```

As usual, unrolling the loop will remove the **if**-test. The $A[i + K]$ value that's prefetched will contain *garbage* – dead data that we know will be overwritten. We perform the prefetch only to avoid the write-miss stall.

If the write-miss policy is *write-around*, then we should prefetch only if we expect the $A[i]$ values to be fetched soon after they are stored.

Summary. Prefetching is applicable when

- The machine has a prefetch instruction, or a non-blocking load instruction that can be used as a prefetch;
- The machine does not dynamically reorder instructions, or the dynamic reorder buffer is smaller than the particular cache latency that we desire to hide; *and*
- The data in question is larger than the cache, or not expected to be already in cache.

I will not describe the algorithm for inserting prefetch instructions in loops, but see the Further Reading section.

21.4 LOOP INTERCHANGE

The most fundamental way of using cache effectively is the reuse of cached data. When nested loops access memory, successive iterations of a loop often reuse the same word, or use adjacent words that occupy the same cache block. If it is the *innermost* loop whose iterations reuse the same values, then there will be many cache hits. But if one of the outer loops reuses a cache block, it may be that execution of the inner loop stomps through the cache so heavily that by the time the next outer-loop iteration executes, the cache block will have been flushed.

Consider the following nested loops, for example.

```
for $i \leftarrow 0$ to $N - 1$
    for $j \leftarrow 0$ to $M - 1$
        for $k \leftarrow 0$ to $P - 1$
            $A[i, j, k] \leftarrow (B[i, j-1, k] + B[i, j, k] + B[i, j+1, k])/3$
```

The value $B[i, j+1, k]$ is reused in the next iteration of the j loop (where its "name" is $B[i, j, k]$), and then is reused again in the iteration after that. But in the meantime, the k loop brings $3P$ elements of the B array, and P elements of the A array, through the cache. Some of these words may very well conflict with $B[i, j+1, k]$, causing a cache miss the next time it is fetched.

The solution in this case is to interchange the j and k loops, putting the j loop innermost:

```
for $i \leftarrow 0$ to $N - 1$
    for $k \leftarrow 0$ to $P - 1$
        for $j \leftarrow 0$ to $M - 1$
            $A[i, j, k] \leftarrow (B[i, j-1, k] + B[i, j, k] + B[i, j+1, k])/3$
```

Now $B[i, j, k]$ will always be a cache hit, and so will $B[i, j-1, k]$.

To see whether interchange is legal for a given pair of loops, we must examine the data-dependence graph of the calculation. We say that iteration (j, k) depends on iteration (j', k') if (j', k') computes values that are used by (j, k) (read-after-write), or stores values that are overwritten by (j, k) (write-after-write), or reads values that are overwritten (write-after-read). If the interchanged loops execute (j', k') before (j, k), *and* there is a dependence, then the computation may yield a different result, and the interchange is illegal.

In the example shown above, there is *no* dependence between any iterations of the nested loops, so interchange is legal.

See the Further Reading section for a discussion of the analysis of dependence relations for array accesses in nested loops.

21.5 BLOCKING

The technique of *blocking* reorders a computation so that all the computations that use one portion of the data are completed before moving on to the next portion. The following nested loop for matrix multiplication, $C = AB$, illustrates the need for blocking:

```
for $i \leftarrow 0$ to $N - 1$
   for $j \leftarrow 0$ to $N - 1$
      for $k \leftarrow 0$ to $N - 1$
         $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
```

If both A and B fit into the cache simultaneously, then the k loop will run without cache misses, and there may be only one cache miss for $C[i, j]$ on each iteration of the j loop.

But suppose the cache is large enough to hold only $2 \cdot c \cdot N$ matrix elements (floating-point numbers), where $1 < c < N$. For example, multiplying 50×50 matrices of 8-byte floats on a machine with an 8-kilobyte cache, $c = 10$. Then *every reference to $B[k, j]$ in the inner loop will be a cache miss*, because – since the last time that particular cell of B was accessed – the entire B matrix will have been marched through the cache, dumping out the "old" values. Thus, each iteration of the inner loop will have a cache miss.

Loop interchange cannot help here, because if the j loop is outermost, then A will suffer cache misses, and if the k loop is outermost, then C will suffer misses.

The solution is to reuse rows of the A matrix and columns of the B matrix while they are still in cache. A $c \times c$ block of the matrix C can be calculated from c rows of A and c columns of B, as follows (see also Figure 21.7):

```
for $i \leftarrow i_0$ to $i_0 + c - 1$
   for $j \leftarrow j_0$ to $j_0 + c - 1$
      for $k \leftarrow 0$ to $N - 1$
         $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
```

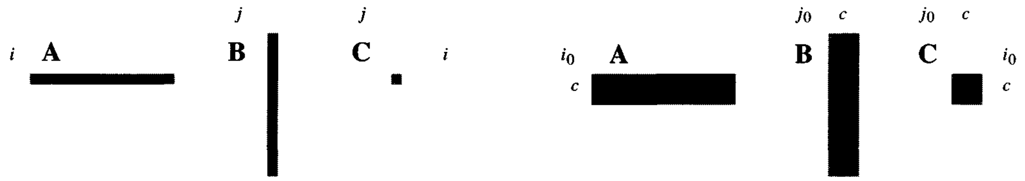

FIGURE 21.7. Matrix multiplication. Each element of C is computed from a row of A and a column of B. With blocking, a $c \times c$ block of the C matrix is computed from a $c \times N$ block of A and a $N \times c$ block of B.

Only $c \cdot N$ elements of A and $c \cdot N$ elements of B are used in this loop, and *each element is used c times*. Thus, at a cost of $2 \cdot c \cdot N$ cache misses to bring this portion of A and B into cache, we are able to compute $c \cdot c \cdot N$ iterations of the inner loop, for a miss rate of $2/c$ misses per iteration.

All that remains is to nest this set of loops inside outer loops that compute each $c \times c$ block of C:

```
for $i_0 \leftarrow 0$ to $N - 1$ by $c$
  for $j_0 \leftarrow 0$ to $N - 1$ by $c$
    for $i \leftarrow i_0$ to $\min(i_0 + c - 1, N - 1)$
      for $j \leftarrow j_0$ to $\min(j_0 + c - 1, N - 1)$
        for $k \leftarrow 0$ to $N - 1$
          $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
```

This optimization is called *blocking* because it computes one block of the iteration space at a time. There are many nested-loop programs on which an optimizing compiler can automatically perform the blocking transformation. Crucial to the situation are loops whose iterations are not data-dependent on each other; in matrix multiplication, the calculation of $C[i, j]$ does not depend on $C[i', j']$, for example.

Scalar replacement. Even though the access to $C[i, j]$ in the matrix-multiply program will almost always hit the cache (since the same word is being used repeatedly in the k loop), we can still bring it up one level in the memory hierarchy – from primary cache into registers! – by the *scalar replacement* optimization. That is, when a particular array element is used as a scalar for repeated computations, we can "cache" it in a register:

```
for $i \leftarrow i_0$ to $i_0 + c - 1$
  for $j \leftarrow j_0$ to $j_0 + c - 1$
    $s \leftarrow C[i, j]$
    for $k \leftarrow 0$ to $N - 1$
      $s \leftarrow s + A[i, k] \cdot B[k, j]$
    $C[i, j] \leftarrow s$
```

This reduces the number of fetches and stores in the innermost loop by a factor of 2.

Blocking at every level of the memory hierarchy. To do blocking optimizations, the compiler must know how big the cache is – this determines the best value of c, the block size. If there are several levels of the memory hierarchy, then blocking can be done at each level. Even the machine's registers should be considered as a level of the memory hierarchy.

Taking again the example of matrix multiply, we suppose there are 32 floating-point registers, and we want to use d of them as a kind of cache. We can rewrite the $c \times c$ loop (of the blocked matrix multiply) as follows:

```
for $i \leftarrow i_0$ to $i_0 + c - 1$
  for $k_0 \leftarrow 0$ to $N - 1$ by $d$
    for $k \leftarrow k_0$ to $k_0 + d - 1$
      $T[k - k_0] \leftarrow A[i, k]$
    for $j \leftarrow j_0$ to $j_0 + c - 1$
      $s \leftarrow C[i, j]$
      for $k \leftarrow k_0$ to $k_0 + d - 1$
        $s \leftarrow s + T[k - k_0] \cdot B[k, j]$
      $C[i, j] \leftarrow s$
```

Unroll and jam. *Loop unrolling must be used for register-level blocking,* since registers cannot be indexed by subscripts. So we unroll the k-loops d times and keep each $T[k]$ in a separate scalar temporary variable (for illustration, I will use $d = 3$, though $d = 25$ would be more realistic):

```
for $i \leftarrow i_0$ to $i_0 + c - 1$
  for $k_0 \leftarrow 0$ to $N - 1$ by 3
    $t_0 \leftarrow A[i, k_0]$;   $t_1 \leftarrow A[i, k_0 + 1]$;   $t_2 \leftarrow A[i, k_0 + 2]$
    for $j \leftarrow j_0$ to $j_0 + c - 1$
      $C[i, j] \leftarrow C[i, j] + t_0 \cdot B[k_0, j] + t_1 \cdot B[k_0 + 1, j] + t_2 \cdot B[k_0 + 2, j]$
```

The register allocator will ensure, of course, that the t_k are kept in registers. Every value of $A[i, k]$ fetched from the cache is used c times; the B values still need to be fetched, so the number of memory accesses in the inner loop goes down by almost a factor of two.

A high-tech compiler would perform – on the same loop! – blocking transformations for the primary cache and for the secondary cache, and scalar replacement and unroll-and-jam for the register level of the memory hierarchy.

21.6 GARBAGE COLLECTION AND THE MEMORY HIERARCHY

Garbage-collected systems have had the reputation as cache-thrashers with bad cache locality: after all, it would appear that a garbage collection touches all of memory in random-access fashion.

But a garbage-collector is really a kind of memory manager, and we can organize it to manage memory for improved locality of reference.

Generations: When generational copying garbage collection is used, the youngest generation (allocation space) should be made to fit inside the secondary cache. Then each memory allocation will be a cache hit, and each youngest-generation garbage collection will operate almost entirely within the cache as well – only the objects promoted to another generation may cause cache-write misses. (Keeping the youngest generation inside the primary cache is impractical, since that cache is usually so small that too-frequent garbage collections would be required.)

Sequential allocation: With copying collection, new objects are allocated from a large contiguous free space, sequentially in order of address. The sequential pattern of stores to initialize these objects is easy for most modern write-buffers to handle.

Few conflicts: The most frequently referenced objects tend to be the newer ones. With sequential allocation of objects in the youngest generations, the *keys* of these newer objects (in a direct-mapped cache) will be all different. Consequently, garbage-collected programs have significantly lower conflict-miss rates than programs that use explicit freeing.

Prefetching for allocation: The sequential initializing stores cause cache-write misses (in the primary cache, which is much smaller than the allocation space) at the rate of one miss per B/W stores, where B is the cache block size and W is the word size. On most modern machines (those with write-validate cache policies) these misses are not costly, because a *write* miss does not cause the processor to wait for any data. But on some machines (those with fetch-on-write or write-around policies) a write miss is costly. One solution is to prefetch the block well in advance of storing into it. This does not require analysis of any loops in the program (like the technique shown in Section 21.3) – instead as the allocator creates a new object at address a, it prefetches word $a+K$. The value K is related to the cache-miss latency and also the frequency of allocation versus other computation, but a value of $K = 100$ should work well in almost all circumstances.

Grouping related objects: If object x points to object y, an algorithm that accesses x will likely access y soon, so it is profitable to put the two objects in the same block. A copying collector using *depth-first* search to traverse the live data will automatically tend to put related objects together; a collector using *breadth-first* search will not. Copying in depth-first order improves cache performance – but only if the cache blocks are larger than the objects.

These cache-locality improvement techniques are all applicable to *copying* collection. Mark-and-sweep collectors, which cannot move the live objects, are less amenable to cache management; but see the Further Reading section.

FURTHER READING

Sites [1992] discusses several kinds of instruction- and data-cache alignment optimizations. Efficient approximation algorithms for the traveling salesman problem (TSP) can be applied to basic-block ordering, to minimize the instruction-fetch penalties for branches [Young et al. 1997].

Mowry et al. [1992] describe an algorithm for inserting prefetch instructions in **for**-loops, taking care not to insert prefetches (which do, after all, have an instruction-issue cost) where the data in question is likely to be in cache already.

The Lisp Machine's garbage collector used depth-first search to group related objects on the same page to minimize page faults [Moon 1984]. Koopman et al. [1992] describe prefetching for a garbage-collected system. Diwan et al. [1994], Reinhold [1994], and Gonçalves and Appel [1995] analyze the cache locality of programs that use copying garbage collection. For mark-sweep collectors, Boehm et al. [1991] suggest that (to improve page-level locality) new objects should not be allocated into mostly full pages containing old objects, and that the sweep phase should be done incrementally so that pages and cache-blocks are "touched" by the sweep just before they'll be allocated by the program.

The techniques for optimizing the memory locality of programs with nested loops have much in common with techniques for parallelizing loops. For example, in a parallel implementation of matrix multiplication, having each processor compute one *row* of the C matrix requires that processor to have N^2 elements of A and N elements of B, or $O(N^2)$ words of interprocessor communication. Instead, each processor should compute one *block* of C (where the block size is $\sqrt{N} \times \sqrt{N}$); then each processor requires $N \cdot \sqrt{N}$ words of A and of B, which is only $O(N^{1.5})$ words of communication. Many of the compilers that use blocking and loop-nest optimizations to generate the most memory-efficient code for uniprocessors are parallelizing compilers – with the parallelization turned off!

To generate good parallel code – or to perform many of the loop optimizations described in this chapter, such as blocking and interchange – it's necessary to analyze how array accesses are data-dependent on each other. Array dependence analysis is beyond the scope of this book, but is covered well by Wolfe [1996].

Callahan et al. [1990] show how to do scalar replacement; Carr and Kennedy [1994] show how to calculate the right amount of unroll-and-jam for a loop based on the characteristics of the target machine.

Wolf and Lam [1991] describe a compiler optimization algorithm that uses blocking, tiling (like blocking but where the tiles can be skewed instead of rectangular), and loop interchange to achieve locality improvements on many kinds of nested loops.

The textbook by Wolfe [1996] covers almost all the techniques described in this chapter, with particular emphasis on automatic parallelization but also with some treatment of improving memory locality.

EXERCISES

***21.1** Write a program in C for multiplying 1000×1000 double-precision floating-point matrices. Run it on your machine and measure the time it takes.

a. Find out the number of floating-point registers on your machine, the size of the primary cache, and the size of the secondary cache.

b. Write a matrix-multiply program that uses blocking transformations at the *secondary cache* level only. Measure its run time.

c. Modify your program to optimize on both levels of cache; measure its run time.

d. Modify the program again to optimize over both levels of cache *and* use registers via unroll-and-jam; view the output of the C compiler to verify that the register allocator is keeping your temporary variables in floating-point registers. Measure the run time.

***21.2** Write a program in C for multiplying 1000 × 1000 double-precision floating-point matrices. Use the C compiler to print out assembly language for your loop. If your machine has a prefetch instruction, or a non-stalling load instruction that can serve as a prefetch, insert prefetch instructions to hide secondary-cache misses. Show what calculations you made to take account of the cache-miss latency. How much faster is your program with prefetching?