

and

$$\left\lceil \frac{\text{Byte address}}{\text{Bytes per block}} \right\rceil \times \text{Bytes per block} + (\text{Bytes per block} - 1)$$

Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lceil \frac{1200}{16} \right\rceil = 75$$

which maps to cache block number $(75 \bmod 64) = 11$. In fact, this block maps all addresses between 1200 and 1215.

Larger blocks exploit spatial locality to lower miss rates. As Figure 5.11 shows, increasing the block size usually decreases the miss rate. The miss rate may go up eventually if the block size becomes a significant fraction of the cache size, because the number of blocks that can be held in the cache will become small, and there will be a great deal of competition for those blocks. As a result, a block will be bumped out of the cache before many of its words are accessed. Stated alternatively, spatial locality among the words in a block decreases with a very large block; consequently, the benefits in the miss rate become smaller.

A more serious problem associated with just increasing the block size is that the cost of a miss increases. The miss penalty is determined by the time required to fetch

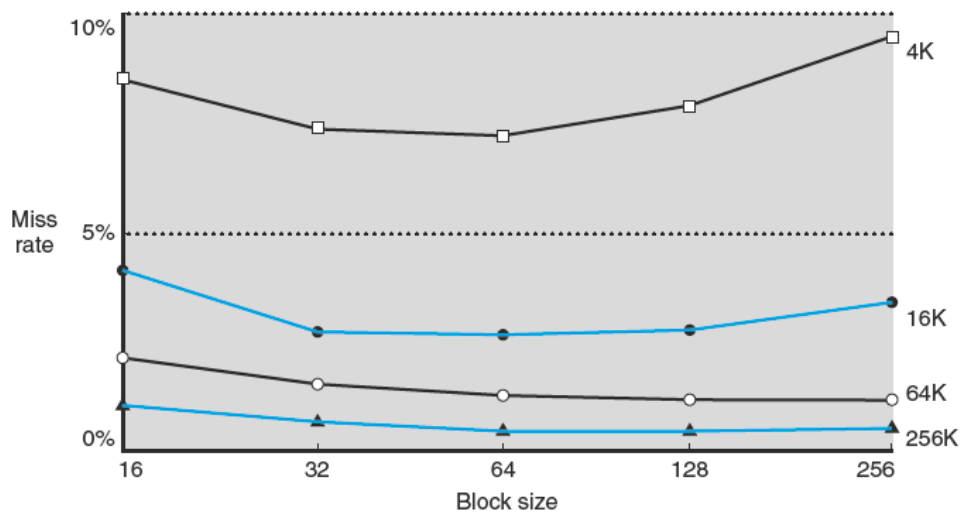


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so this data is based on SPEC92.

the block from the next lower level of the hierarchy and load it into the cache. The time to fetch the block has two parts: the latency to the first word and the transfer time for the rest of the block. Clearly, unless we change the memory system, the transfer time—and hence the miss penalty—will likely increase as the block size increases. Furthermore, the improvement in the miss rate starts to decrease as the blocks become larger. The result is that the increase in the miss penalty overwhelms the decrease in the miss rate for blocks that are too large, and cache performance thus decreases. Of course, if we design the memory to transfer larger blocks more efficiently, we can increase the block size and obtain further improvements in cache performance. We discuss this topic in the next section.

Elaboration: Although it is hard to do anything about the longer latency component of the miss penalty for large blocks, we may be able to hide some of the transfer time so that the miss penalty is effectively smaller. The simplest method for doing this, called *early restart*, is simply to resume execution as soon as the requested word of the block is returned, rather than wait for the entire block. Many processors use this technique for instruction access, where it works best. Instruction accesses are largely sequential, so if the memory system can deliver a word every clock cycle, the processor may be able to restart operation when the requested word is returned, with the memory system delivering new instruction words just in time. This technique is usually less effective for data caches because it is likely that the words will be requested from the block in a less predictable way, and the probability that the processor will need another word from a different cache block before the transfer completes is high. If the processor cannot access the data cache because a transfer is ongoing, then it must stall.

An even more sophisticated scheme is to organize the memory so that the requested word is transferred from the memory to the cache first. The remainder of the block is then transferred, starting with the address after the requested word and wrapping around to the beginning of the block. This technique, called *requested word first* or *critical word first*, can be slightly faster than early restart, but it is limited by the same properties that limit early restart.

Handling Cache Misses

Before we look at the cache of a real system, let's see how the control unit deals with **cache misses**. (We describe a cache controller in detail in Section 5.9). The control unit must detect a miss and process the miss by fetching the requested data from memory (or, as we shall see, a lower-level cache). If the cache reports a hit, the computer continues using the data as if nothing happened.

Modifying the control of a processor to handle a hit is trivial; misses, however, require some extra work. The cache miss handling is done in collaboration with the processor control unit and with a separate controller that initiates the memory access and refills the cache. The processing of a cache miss creates a pipeline stall (Chapter 4) as opposed to an interrupt, which would require saving the state of all registers. For a cache miss, we can stall the entire processor, essentially freezing the contents of the temporary and programmer-visible registers, while we wait

cache miss A request for data from the cache that cannot be filled because the data is not present in the cache.